TOOL SUPPORT FOR INTRODUCTORY SOFTWARE ENGINEERING EDUCATION

by

Andrew Patterson

A thesis submitted in fulfilment of the
requirements for the degree of

Doctor of Philosophy

School of Computer Science and Software Engineering
Monash University

September 2002

This thesis has not been submitted for the award of any degree or diploma in any other tertiary institution. No other person or person's work has been used without due acknowledgement.

Andrew Patterson

September 2002

ABSTRACT

The teaching of software engineering in introductory courses can be supported through the use of software tools. In this thesis, we identify tasks that students may be expected to perform in an introductory software engineering course and evaluate the tool support that is currently available for these tasks. We then concentrate on the areas of refactoring and testing as areas that are deficient in suitable tools for first year students. For these two areas we propose extensions to the BlueJ development environment that are designed for first year students.

A design for a refactoring module is proposed that adds method-centric refactoring functionality, where methods become first class user interface objects supporting standard refactorings such as rename, move and extract.

A testing module is designed and implemented that integrates the existing object interaction facilities of BlueJ with the JUnit testing framework to allow the automatic creation of test cases based on actual user interaction.

ACKNOWLEDGMENTS

When you spend a long time at university working on a project, many people contribute and help you out along the way. I have chosen the list form, as it is probably most efficient, so thanks go out:

- First and foremost to my supervisors John Rosenberg and Michael Kölling for all their help, support and insight over the years. Despite both having heavy workloads during the final months of my thesis preparation, they were always willing to promptly read and comment on chapters. Also to Bruce Quig, the other member of the BlueJ research group, for all the helpful comments during meetings and for reading and commenting on my final draft;

- To my housemates Akiko, Brad, Gurdeesh, Gajan and Toni for putting up with my nocturnal behaviour;

- To everyone from Howitt Hall, who may not have contributed to the quick completion of my thesis, but who were great to live with when I first arrived in Melbourne knowing no one;

- To my family, who have had to put up with questions like "hasn't your son finished university yet" for the past 6 years.

TABLE OF CONTENTS

INTRODUCTION 13
INTRODUCTORY SOFTWARE ENGINEERING EDUCATION 17
2.1 The Software Engineering Body of Knowledge (SWE-BOK) 17
2.1.1 Computing Fundamentals 19
2.1.2 Software Domains 20
2.1.3 Software Management 20
2.1.4 Software Product Engineering 21
2.2 Software Product Engineering – further analysis 21
2.2.1 Software Requirements Engineering (KA 2.1) 21
2.2.2 Software Design (KA 2.2) 22
2.2.3 Software Coding (KA 2.3) 22
2.2.4 Software Testing (KA 2.4) 23
2.2.5 Software Operation and Maintenance (KA 2.5) 23
2.3 Software Product Engineering - tasks 24
2.3.1 Design a system and draw design diagrams 25
2.3.2 Create graphical user interfaces 26
2.3.3 Search and create documentation 27
2.3.4 Enter and edit programs 29
2.3.5 Browse class libraries 30
2.3.5.1 Visualising relationships 31
2.3.5.2 Navigating relationships 32
2.3.5.3 Smalltalk 33
2.3.5.4 EiffelStudio 35
2.3.6 Build programs 37
2.3.7 Implement and execute test cases 38
2.3.7.1 Class level 39

2.3.7.2 Object level 40
2.3.8 Run and debug applications 40
2.3.8.1 Smalltalk systems 41
2.3.8.2 Self Environments 41
2.3.8.3 Object Class Browser 42
2.3.8.4 DrJava 42
2.3.9 Refactor code 44
2.3.9.1 Smalltalk Refactoring Browser 45
2.3.9.2 IntelliJ IDEA 46
2.3.10 Integrate external resources 47
2.4 Summary and motivation 47
REFACTORING 49
3.1 Introduction 49
3.2 Why refactor in first year? 51
3.3 What refactorings are appropriate? 52
3.3.1 Changes local to a method fragment 54
3.3.2 Changes to a method signature 55
3.3.3 Changes to a class structure 56
3.3.3.1 Move operations 56
3.3.3.2 Extract operations 57
3.3.3.3 Inheritance structure operations 59
3.3.4 Changes to the design 60
3.3.5 Summary 61
3.4 Current tool support for refactoring 63
3.5 A design for an introductory refactoring tool 66
3.5.1 Methods as user interface objects 67
3.5.2 Classes as user interface objects 67
3.5.3 System wide undo 68
3.5.4 Summary 69
TESTING 71
4.1 Why test? 71

4.2 Testing in education ... 73
4.2.1 The early software engineering approach ... 74
4.2.2 Early testing ... 76
4.3 Testing techniques for students ... 78
4.4 Current tool support for testing ... 83
4.4.1 Symbolic debuggers ... 83
4.4.2 Unit testing with JUnit ... 84
4.4.3 TestMentor ... 87
4.4.3.1 Construction of test assets ... 87
4.4.3.2 Construction of test stubs ... 89
4.4.3.3 Validation ... 90
4.4.3.4 Summary ... 91
4.4.4 BlueJ ... 92
4.5 Summary ... 93
DESIGN OF TESTING SUPPORT IN AN EDUCATIONAL INTEGRATED DEVELOPMENT ENVIRONMENT ... 95
5.1 Blue ... 95
5.2 BlueJ ... 96
5.2.1 UML style class diagrams ... 97
5.2.2 Direct object interaction ... 98
5.2.3 Object inspection ... 99
5.2.4 Integrated debugger ... 99
5.2.5 Javadoc generation ... 100
5.3 Introduction to testing in BlueJ ... 101
5.4 Testing overview ... 101
5.5 Conventional testing walk through ... 103
5.5.1 Recording of Ad-Hoc Test Interaction ... 104
5.5.2 Constructing the test class ... 104
5.5.3 Creating a test method ... 106
5.5.4 Asserting results ... 107
5.5.5 Run All ... 108

5.5.6 Dealing with arrays 109
5.5.7 Testing using standard Java classes 111
5.5.8 Sharing test objects 112
5.5.9 Creation of a test fixture 113
5.5.10 Restoring a test fixture 114
5.5.11 Extending a test fixture 116
5.5.12 Silent compilation 116
5.5.13 Tests created outside of BlueJ 116
5.5.14 Run individual tests 117
5.5.15 Testing exceptions 118
5.5.16 Free form assertions 119
5.5.17 Further ideas 119
5.6 Test driven development 120
5.6.1 Walkthrough 120
5.6.2 Summary 123
IMPLEMENTATION 125
6.1 Implementation environment 125
6.2 High level overview 125
6.3 Constructing test fixtures 128
6.3.1 Java Object Serialization (JOS) 129
6.3.2 JSX 132
6.3.3 XMLEncoder and XMLDecoder 133
6.4 Creation of the text fixture and test methods 135
6.5 Architectural changes to support testing 139
6.6 Implementing the Runner 141
6.7 Summary 142
STATUS AND FUTURE WORK 143
7.1 Status 143
7.2 Usability Study of the Unit Testing Extension 144
7.2.1 Experimental Procedure 144
7.2.2 Results 146

7.2.3 Discussion 149
7.3 Extended functionality 149
7.3.1 Extending test methods 149
7.3.2 Support multiple test cases associated with a single target class 149
7.3.3 Test coverage analysis 150
7.4 Further tool support for introductory software engineering education 150
CONCLUSION 153
GLOSSARY 156
REFERENCES 158

LIST OF FIGURES

Figure 1 - The IBM Visual Age for Java search dialog can search based on a semantic understanding of the source code.....29
Figure 2 – An example of IntelliJ QuickInfo showing the popup display that occurs when the editor caret is placed on a method call.....30
Figure 3 - A Smalltalk Browser.....33
Figure 4 – A "Development Window" in EiffelStudio targeted on the STRING class (reproduced with permission from [Meyer2001 page 20]).....35
Figure 5 – The "class" view (reproduced with permission from [Meyer2001 page 31]).....36
Figure 6 – The "cluster" view in EiffelStudio showing the empty MY_CLUSTER that has just been created as a child of the ROOT_CLUSTER (reproduced with permission from [Meyer2001 page 21]).....38
Figure 7 – DrJava showing object interaction being performed in the lower panel.....43
Figure 8 - Performing a refactoring with IntelliJ IDEA.....46
Figure 9 – The refactoring menu in IntelliJ IDEA.....64
Figure 10 – The context in which refactorings are appropriate.....65
Figure 11 – The popup menu attached to a method in the editor.....66
Figure 12 – The popup menu attached to a class in the editor.....67
Figure 13 – The History/Undo window in Adobe Photoshop.....69
Figure 14 – Two orthogonal classifications of testing.....73
Figure 15 – A sample of test code written using the JUnit framework.....85
Figure 16 – The SwingRunner showing the result of the EmailTest.....87

Figure 17 – Construction of "steps" in Test Mentor (reproduced with permission from [Silvermark2002 page 92]). 88
Figure 18 – Recording object interaction with Test Mentor (reproduced with permission from [Silvermark2002 page 84]). 90
Figure 19 – The main BlueJ window showing the UML style class diagram and objects on the object bench. 96
Figure 20 – The popup menu of a class in BlueJ 97
Figure 21 – Parameter passing when constructing an object in BlueJ 98
Figure 22 – Inspecting an object. 99
Figure 23 – The BlueJ debugger. 100
Figure 24 – The BlueJ system showing the addition of the unit testing functionality. 102
Figure 25 – The popup menu for creating a new test class. 104
Figure 26 – The popup menu for creating a test method. 106
Figure 27 – The result and assertion dialog. 107
Figure 28 – The unit test source of a method created through BlueJ interactions in the ParserTest class. 108
Figure 29 – The dialog showing the result of running three tests. 109
Figure 30 – The result and assertion dialog for an array. 110
Figure 31 – The unit test source of a basic method created through BlueJ interactions in the ParserTest class. 111
Figure 32 – A `java.io.StringReader` object on the object bench. The popup menu shows the method calls which can be made on the object. 112
Figure 33 – The unit test source for a method created using the Java StringReader class. 113
Figure 34 – The method call dialog executing Room's setExits() method. 114
Figure 35 – The result and assertion dialog when the object returned is already on the object bench. 115
Figure 36 – The unit test source for a test method generated when an exception is caught. 118

Figure 37 – The free form assertion dialog. 119
Figure 38 – The unit test source for a TDD method in TransporterRoomTest.121
Figure 39 – Editing the TransporterRoomTest in the BlueJ editor. 122
Figure 40 – A simplified view of the BlueJ system. 126
Figure 41 – A GUI component serialized to XML using XMLEncoder and how the component would look as Java code. 133
Figure 42 – A graph recording the transitive closure of all operations on the object A. 135
Figure 43 – The objects on an object bench recorded as a sequence of operations. 137
Figure 44 – Using the scoping rules of Java to allow the "result" variable to be reused within a method. 139

1 Chapter 1

INTRODUCTION

The philosophy of what to teach in introductory computer programming courses has changed markedly over the past thirty years. Scanning the early ACM SIGCSE proceedings, one sees a discipline struggling to establish itself, as new computer science departments formed from the existing mathematics and physics departments in universities. Much of what was taught reflected the backgrounds of the departments, leading to an emphasis on numerical methods, computational complexity theory and proofs, alongside courses in the rapidly advancing areas of operating systems and computer hardware [ACM1968]. However, some prescient insights were made that are now starting to be reflected in the modern approach to teaching introductory computer programming:

"In the design of this course I have taken a much broader view [of the meaning of software engineering]. I take the view that programming is taught in our basic courses as a solo activity. Such courses teach programming techniques that are suitable for use by a single person constructing a program which will not be touched by other people. In contrast, I feel that the essential characteristic of a software engineering task is that many people will be involved with the product. Either several people will cooperate in producing it, or it will be used or modified by persons other than the original writer"

– David Parnas "A Course on Software Engineering Techniques" ACM SIGCSE 1971

As the decades went on, computing curricula evolved along with the discipline itself. The 80's saw a shift to more experimental work as computing power became increasingly accessible [ACM1979]. Into the 90's, we saw an emphasis on algorithms and data structures, although more recognition was made of the importance of software engineering skills [ACM1991].

The major change in teaching now as we enter the 00's is the change to object orientation and the emergence of software engineering skills in introductory courses. The shift from teaching procedural programming languages to object-oriented programming languages, and the emergence of software engineering as a separate discipline, has meant that other skills now need to be introduced to first year students to best support the new paradigms.

The introduction of object-oriented programming has *mandated* some changes. For instance, code reuse is now an integral part of programming and the use of class libraries is a required skill for any competent programmer. Time must now be set aside for the introduction of these standard class libraries. In fact, some of the time that once may have been spent learning how to implement certain data structures may now need to be spent learning how to use standard implementations of these data structures.

Other changes have been *enabled* through the introduction of object-orientation. Modern graphical user interface toolkits and testing frameworks have a simplicity and clarity now that was not obtainable in the world of procedural languages. This clarity now makes it feasible to introduce these topics to students in an introductory course.

The software engineering skills that are being introduced flow on from the change to object-orientation. Reasonably complex programs can be constructed and presented to students with an appropriate level of modularity, such that they are only required to modify one class, yet perhaps understand the design of three or four other classes. It is finally possible to have students work on "large" programs without overwhelming them with complexity, and yet not requiring too much hand waving to explain away the advanced classes.

Other important software engineering skills such as teamwork are better supported because there can be a much clearer separation of concerns between component groups within the team. Similarly, testing is more effective with object-oriented

code because it is easier to isolate units of the overall system and test these individually.

How then have the tools we use in introductory teaching changed with recent curriculum developments? It is true that integrated development environments have become larger and more functionality has been added, yet very little has changed about the fundamental way in which they work. They are still very much oriented around the concepts and abstractions of procedural programming. We contend that some aspects of teaching object-orientation and software engineering are not well supported by tools that are currently available for introductory students. The purpose of this thesis is to investigate this claim and to propose remedies for some deficient areas of software engineering education by designing and implementing extensions to an integrated development environment.

Chapter 2 provides background to the software engineering discipline and develops a set of software engineering tasks through which existing software tools are evaluated. We identify two areas where existing tools are deficient, namely refactoring and testing.

In chapter 3 we discuss the area of refactoring and the importance of support for refactoring from software tools. A design of a refactoring tool especially designed for introductory students is presented.

Chapter 4 discusses the area of testing and looks at approaches to teaching testing in first year courses. From this analysis, we lead into chapter 5 in which we describe the design of a new and novel tool that integrates support for testing within the BlueJ programming development environment. In chapter 6 we then discuss the implementation issues arising from the addition of testing support to BlueJ.

Chapter 7 discusses the status of the work that has already been performed and contains some ideas for future work. Chapter 8 provides a conclusion and summary of this thesis and its contributions.

2 Chapter 2

INTRODUCTORY SOFTWARE ENGINEERING EDUCATION

The aim of this chapter is to introduce approaches used for introductory software engineering education. We begin by examining the software engineering body of knowledge, and identifying significant educational approaches to teaching in each knowledge category. In particular we look at the part that software tools play in the teaching of these concepts. The latter part of the chapter concentrates on the software product engineering category since this an area which we feel can benefit a great deal from the use of software tools. For topics in this category we look at the tasks that students are required to perform, the support software development tools give them for performing these tasks, and the suitability of these tools for introductory students.

2.1 The Software Engineering Body of Knowledge (SWE-BOK)

It is the contention of this thesis that many areas in software engineering can be introduced to students in first year with appropriate support from software tools. In fact, we believe that the move to object-oriented programming in first year has made tool support essential. An evaluation of the strengths and weaknesses of tools, however, cannot be performed against generalised notions such as design, implementation or testing. A better approach is to identify practical tasks that students perform in first year and evaluate tools against these concrete tasks.

In attempting to identify some of these practical tasks, we will start by looking at recent efforts to establish a body of knowledge[1] for software engineering. This body of knowledge will lead us through the key concepts of software engineering and allow us to discover a set of tasks that can be used for evaluating tool support.

[1] A body of knowledge is an attempt to codify and categorise the nature and content of a discipline.

The Software Engineering Body of Knowledge (SWE-BOK) [Hilburn1999] is an effort of the SECC (Software Engineering Coordinating Committee), a joint committee of the Association for Computing Machinery (ACM) and the IEEE Computer Society (IEEE-CS). The development of the SWE-BOK was motivated by "the lack of a clear and comprehensive understanding of the nature and content of the software engineering profession" [Hilburn1999 page 1]. The SWE-BOK sets out to define a hierarchy of concepts in software engineering. At the top level are the four Knowledge Categories (KC). These are:

1. Computing Fundamentals
2. Software Product Engineering
3. Software Management
4. Software Domains

Within each KC are Knowledge Areas (KA). Finally, within each KA are Knowledge Units (KU) which define each individual atomic concept.

In the following sections we will look at each category and identify methodologies for teaching in the area at an introductory level. Of particular interest will be the support provided by software tools for this teaching. The support that tools provide can be twofold; firstly, some software engineering concepts are naturally tool based and learning a tool is a necessary part of mastering the concept. For instance, using a compiler, or the use of an integrated development environment fall into this category. The second form of support that a tool can give is in reinforcing some of the concepts of software engineering that may not normally involve a specialist tool. For instance, a tool can be used to visualise an algorithm which helps the student learn how the algorithm works.

It should be noted that we will deal with knowledge categories out of sequence compared to their ordering in the SWE-BOK. We will leave the Software Product Engineering category until last as it contains the majority of the topics that are of interest when looking at tool support.

2.1.1 Computing Fundamentals

This knowledge category covers the fundamental concepts of computing such as algorithms and data structures [Ginat2001], computer architecture, mathematical foundations, programming languages and operating systems [Hughes2000].

There are two main schools of thought as to how a first year computing course should be structured. A "depth-first" course emphasises one programming language and concentrates on teaching this for the introductory course. A "breadth-first" approach introduces selected topics from each of the computing fundamental knowledge areas, with less of an emphasis on programming languages. Both approaches are equally valid and as the use of tools is not affected by the choice of course structure no more will be said about this topic [McKim1996].

Another contentious question in computing education is whether to teach procedural or object-oriented programming in the introductory course. Much has been written about this topic over the last ten years but it is generally recognised that an object-oriented approach is preferable given the availability now of suitable object-oriented languages for teaching (e.g. Java, C#) [Reges2002]. A review of this debate would run into many thousands of words so we will assume in this thesis the validity of the object-oriented approach. As we will see, the change to object-oriented programming has raised many issues that make adequate tool support even more crucial.

Unlike some of the other knowledge categories which we will look at later, the concepts in this knowledge category do not require the use of software tools in order to be taught effectively. However, there has been some work in this area with tools that are pedagogically designed. These range from Nachos [Christopher1993], a simulated operating system for operating system experimentation, to SPIM [Larus1997], a MIPS R2000 simulator to help learn computer architecture.

Some have proposed tools to help in algorithm visualisation [Naps2000] although the usefulness of algorithm visualisation tools without interactive feedback has been questioned [Stasko1993] [Jarc2000].

2.1.2 Software Domains

This knowledge category specifies software domains that involve the application or utilisation of knowledge from computing and software engineering. The domains include artificial intelligence, database systems, human-computer interaction and real time systems. These domains are generally taught in specialised courses in later years, not in an introductory programming course. Because the domains are so large and quite specialised, we will not discuss them any further and instead refer the interested reader to the major computer science educational conferences, SIGCSE and ITiCSE for examples of tools and methodologies.

2.1.3 Software Management

This knowledge category specifies the domains that involve managing a project and managing the people working on the project. The knowledge areas of this category are project management, risk management, quality assurance, configuration management and process management. The skills involved with many of these areas are very much people skills. Whilst extremely important, they are generally not taught formally at an introductory level. Instead, students gain experience at project management by being involved with group work and perhaps by being asked to develop on-paper testing plans.

There has been some work on introducing software management into introductory curricula. The use of the team software process (TSP) and personal software process (PSP) [Hilburn1997] has students evaluating their own project's success and the success of the process of creating the project. This teaching approach is discussed in more detail in section 4.2.1. Web-based data entry tools have been used for data collection in one implementation of the PSP [Postema2000].

The tools for software management are specialised project management tools that allow the construction of various charts and timelines. Examples of some mainstream software in this category are Microsoft Project and Rational Concepts. Because of the professional nature of these products they are not suitable for students at an introductory level [McDonald2001].

2.1.4 Software Product Engineering

Traditionally, software product engineering and in particular, coding and testing, are taught before software management because it is generally accepted that software management cannot be understood reasonably without any experience in actually engineering software. This makes it a good candidate for potential introductory tool support.

The software product engineering category is also an interesting category for investigating tool support in introductory teaching because many of the topics consist of tasks that require tools in order to perform them effectively.

This knowledge category is split into five knowledge areas: requirements engineering, design, coding, testing and operation and maintenance. These in turn are split into many knowledge units. In the following section, when each knowledge area and knowledge unit is discussed it will be accompanied by its KA or KU number allowing it to be referenced in the SWE-BOK document. For many of the units we will identify a task that either requires or can be aided with the use of a software tool. These tasks will be highlighted in **bold** and will be discussed in more depth from section 2.3 onwards.

2.2 Software Product Engineering – further analysis

2.2.1 Software Requirements Engineering (KA 2.1)

This area looks at techniques for "establishing a common understanding of the requirements to be addressed by a software product" [Hilburn1999 page 17]. Requirements engineering can be done in an ad-hoc manner in introductory courses (the specifications of an assignment may be deliberately obtuse or incomplete,

requiring the students to ask the teacher for more details) but a formal treatment of it will usually be left for later year courses.

2.2.2 Software Design (KA 2.2)

This area is about the formation of a plan detailing how the requirements for a software product are to be met. Most of the tasks in this area are paper/whiteboard tasks and do not need any support from computer tools. However, there are two units of the area which can use tool support.

The Abstract Specification (KU 2.2.2) unit involves learning how to specify object-oriented designs, structured designs and real-time systems designs. The tasks that introductory students may be required to invoke the use of various design methodologies to **design a system**, and then **draw designs** in the form of class diagrams and sequence diagrams.

The Interface Design (KU 2.2.3) unit is concerned with the design of the boundary between the software system and the user. The task of designing these interfaces may be aided by software tools such as a GUI builder. A GUI builder can be used to **create graphical user interfaces** quickly and easily. Some paper tasks can involve the evaluation and comparison of existing interfaces.

2.2.3 Software Coding (KA 2.3)

This area deals with the construction of software to meet the criteria specified in a design. The Code Implementation (KU 2.3.1) unit is concerned with knowing about various programming languages and programming paradigms and how to use source code development tools. Tools are used for **entering and editing programs** and the student needs to learn the use of the system's build tools in order to **build programs**.

The Code Reuse (KU 2.3.2) unit is concerned with using existing code and libraries of code in programs. It also deals with techniques for developing reusable code. The tasks that students will be faced with in this unit are reading and understanding

class interface definitions and **browsing** large class libraries in order to find classes that are suitable for reuse. Tools which help visualise the library of classes and navigate through these libraries are of use here.

The Code Standards and Documentation (KU 2.3.3) unit is about documentation standards for software and the development of internal and external program documentation. Tasks that students may need to perform are **searching the documentation** to find out more about a class or method given its name or method signature. They also may want to search the documentation based on a class keywords. Another related task is the task of **creating the documentation**.

2.2.4 Software Testing (KA 2.4)

This area deals with establishing the correctness of a program. It involves testing of all scopes, from unit testing (KA 2.4.1) through to acceptance testing (KA 2.4.5). Not all forms of testing will be appropriate for introductory students. For instance, because of the small scale of the software projects that they may work on, the difference between integration (KA 2.4.2) and system testing (KA 2.4.3) is probably not great, and this distinction may not be worth emphasising. Some other elements of testing such as performance (KA 2.4.4) and installation testing (KA 2.4.6) are also topics that are best left to more advanced courses.

For each of these testing scopes, students will need to develop test plans. This is an exercise that can be done on paper. The tasks of **implementing test cases** and **executing test cases** are tasks that can be aided with software tool support. As part of the day to day activities of developing software, students also may test their code by **running and debugging** their programs.

2.2.5 Software Operation and Maintenance (KA 2.5)

This area concerns the "methods, process, and techniques that support the ability of a software system to change, evolve and survive" [Hilburn1999 page 18]. The unit Software Maintenance Operations (KA 2.5.2) deals with all aspects of maintenance such as fixing bugs, **refactoring code** to make it more maintainable, and adapting

software to work on other platforms. Whilst the scale and time span of projects undertaken in an introductory course is quite limited, it is possible to introduce some of these aspects in the course of the student's day to day programming. With the improved tool support for refactoring and restructuring now available there is more potential for introducing these tasks to students at an early stage.

The Software Installation and Operation (KA 2.5.1) unit deals with techniques for installing software products and operation of products. One aspect of this of interest to introductory students is **integrating external resources** into development environments. This can occur when a teacher has provided an external resource (such as a class library) that is needed for a project. Whilst some laboratories will automatically configure the student's development environment for new external resources, students may need to deal with integrating these resources on their home computing platform.

2.3 Software Product Engineering - tasks

In the previous section we have looked at all the knowledge units that make up the Software Product Engineering category and identified a set of tasks that students may be required to perform. We have attempted to identify the tasks from the product engineering domain which we feel can most benefit from tool support.

The identification of tasks gives us a framework for evaluation of the usefulness of software tools that is more "testable" than merely looking at a list of software engineering topics. The task list is certainly not definitive though; there may be some tasks that could be added or tasks that could be split into other tasks. However, it gives us a good starting point for our evaluation of software tools for introductory software engineering.

As a summary, the list of tasks is presented in the following table. The assignment to general categories (i.e. design, coding, etc) is not an attempt to classify tasks and thereby present a taxonomy of tasks, it is merely to identify the knowledge area

from which the task was first identified. Some tasks such as debugging could easily be placed in multiple categories.

Design	Design a system and draw design diagrams (see section 2.3.1)
	Create graphical user interfaces (see section 2.3.2)
Coding	Search and create documentation (see section 2.3.3)
	Enter and edit programs (see section 2.3.4)
	Browse class libraries (see section 2.3.5)
	Build programs (see section 2.3.6)
Testing	Implement and execute test cases (see section 2.3.7)
	Run and debug applications (see section 2.3.8)
Maintenance	Refactor code (see section 2.3.9)
Operations	Integrate external resources (see section 2.3.10)

In the following sections we examine each of these tasks in more detail and identify potential tools to assist with their introduction.

2.3.1 Design a system and draw design diagrams

Designing a system using object-oriented design techniques can be done in front of a computer or can be done as a paper based task. The use of CRC cards can allow groups to design systems without needing to use a computer [Beck1989]. Designing with CRC cards involves identifying (C)lasses, their (R)esponsibilities and their (C)ollaborators in the system. These are documented on small index sized cards, with the class name at the top, responsibilities listed down the left and collaborators listed on the right. Execution scenarios are used to discover classes needed and refine the design. When an execution scenario requires a responsibility not already covered, either a new class is created, or extra responsibilities are added to existing classes. If adding a responsibility causes a class to become too large, it is split and its responsibilities are copied over to the new classes. Some work has been done on evaluating CRC card design in the teaching of an introductory subject [Johansson2001].

Students may also be encouraged to construct designs on a computer using UML diagramming tools such as ArgoUML or Rational Rose [Boggs1999]. A problem with using these tools in introductory education is that they are designed for professional software engineers and hence often contain concepts and functionality that is inappropriate for introductory students. One aspect of this advanced functionality is round trip engineering, which allows UML models to be converted into code and existing code to be converted back into UML models. This functionality can lead students into constructing large, complex and inappropriate systems because the generation of the code skeletons is automated (and the complexity only becomes a problem when the student attempts to fill in the skeleton).

A more appropriate tool for introductory students may be a general purpose diagramming tool such as Visio [Eaton2001] which can be used to construct simple UML diagrams.

2.3.2 Create graphical user interfaces

The use of graphical user interfaces (GUIs) in introductory courses is becoming increasingly popular. The use of GUIs has two benefits for students:

- most modern programming involves GUIs so it is a useful skill to be introduced to [Culwin1999]; and
- graphical interfaces are appealing to students and keep them interested in a project [Mutchler1996].

There are those who oppose the introduction of GUIs and event driven programming. Some claim that there is not enough time to deal with user interface construction in an introductory course without skimping on other more fundamental areas. Others claim that the very nature of event driven programming is too complex to be introduced to beginning students. An excellent summary of these issues can be found in [Bruce2001].

Many program development environments come with a graphical user interface builder tool. This tool allows the rapid construction of interfaces using simple drag and drop of the interface components. As with UML tools, it is important that students do not get carried away constructing large complex user interfaces, and then not have time to actually implement the project's functionality.

A way to deal with the complexity of GUI builders and the event driven programming model may be to use GUI toolkits [Rasala2000] [Rasala2001]. A toolkit provides a simple set of classes that allows GUI's to be built without many of the complicated aspects of GUI programming.

2.3.3 Search and create documentation

The task of searching for a class with given attributes is a very common one for programmers. In particular, the inexperience that students have with a language means that they are often not able to remember the types of arguments for a method call or the list of methods available in a class. In some cases a student may need to search for classes which contain a keyword in the documentation or which match some natural language query. For all these cases an effective searching tool is required.

The searching task can be broken down into two subtasks. Firstly, how (and what) information about the classes in the project is entered into the project's documentation. Secondly, what is the method through which this information is queried?

Information about the classes in a project can come from varied sources.

(1) Information added manually.

When a student adds a class to a project they may also be required to add documentation such as keywords or descriptions of functionality.

(2) Information derived automatically from the classes' source.

At some point in the development process tools may be run which automatically generate information from the classes in the project. This may be embedded documentation (for example `javadoc`, which is a standard for embedding documentation in the comments of Java programs [Gosling1999] or the concept of literate programs [Ramsey1994]), or maybe an intelligent automated analysis tool.

(3) Information derived from usage patterns.

It is possible for a system to collate information about the frequency or type of usage which classes get and store this information. A simple example would be to record the most popular classes by various criteria such as number of times used as a superclass or number of instantiations.

The Melmoth system [Hitchens1994] suggests the use of a thesaurus to expand the usefulness of keyword searches although its effectiveness is impeded by the difficulty of creating an effective thesaurus.

The querying methods described above are most effective when used in conjunction with the browsing techniques detailed in section 2.3.5. Queries may not find the ideal class that the student desired, but they are often a very good starting point for a browsing process.

Tool support for searching is generally very good. However, there are only a few IDEs that support a uniform mechanism for searching of all information about a class. For instance, Visual Age for Java (see Figure 1) has a comprehensive search dialog that allows searching with wildcards within the names of types, methods, fields and constructors. Visual Age uses the semantic understanding of the source code built during its compile phase to allow searching of these specific language structures. However, this search dialog cannot search the associated class documentation, nor go directly to the class documentation once a class is found, despite the class documentation being available within the IDE (and indeed searchable through a separate mechanism).

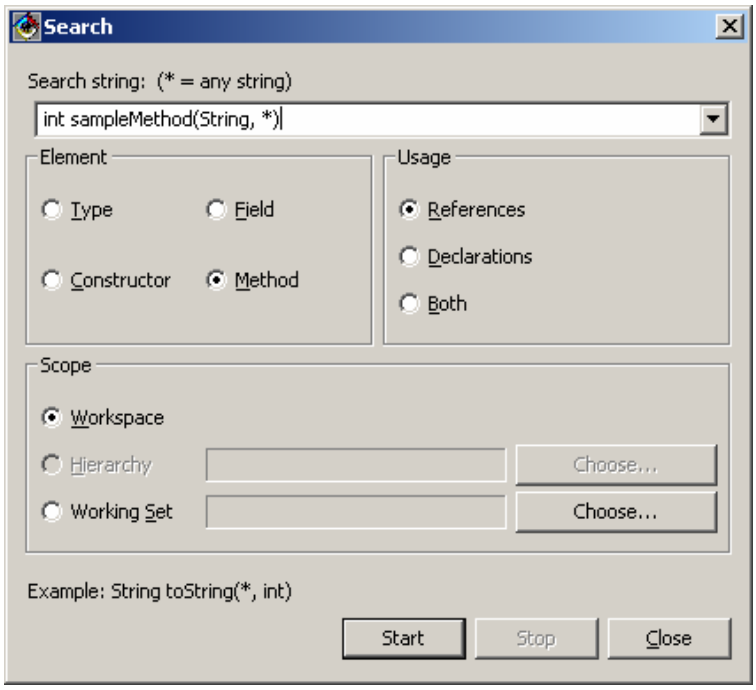

Figure 1 - The IBM Visual Age for Java search dialog can search based on a semantic understanding of the source code.

2.3.4 Enter and edit programs

An editor is the tool that most first year students will spend a significant amount of their time using. Luckily, student's familiarity with word processors such as Microsoft Word means that learning the use of text editors in the preparation of their program source is not a significant hurdle.

Text editors for program development are often augmented with syntax highlighting, where keywords of the programming language are shown in alternative colours. Many also implement bracket matching which shows the corresponding opposing bracket whenever the cursor is over a bracket character. The colour highlighting helps students visually distinguish various parts of their source and the bracket matching helps them track down unmatched brackets (a frustrating error for new programmers).

A program development environment that maintains meta-information about the symbols of a source file (normally by keeping a parse-tree after compilation and

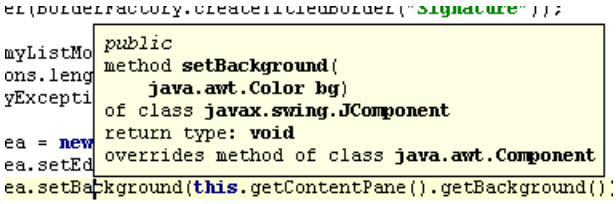

Figure 2 – An example of IntelliJ QuickInfo showing the popup display that occurs when the editor caret is placed on a method call.

linking the resolved symbols back to their location in the source) can offer some additional searching functionality to programmers as they edit programs. At the point where a programmer types in a method name, a search is automatically made for the details of that method or class. The result of this search is then discretely displayed to the programmer at the point where they are editing.

An example of this is IntelliJ's QuickInfo (see Figure 2) that displays metadata of a method and its class whenever the editor caret is placed on a method call in the source text. Both Microsoft Visual Studio and Borland Delphi have similar implicit searching of methods to help programmers complete method calls. This functionality helps students determine the correct parameters to pass to methods and the names of the methods available in a class.

Some types of editors aim to assist students by only allowing syntactically correct programs to be typed into the editor [Khwaja1993]. These syntax-directed editors may be of assistance to students early on but they do not provide an easy progression to the good editing practices required for more mainstream editors of professional development environments. This may be the reason why they are rarely used these days, even in introductory environments.

2.3.5 Browse class libraries

While the concept of libraries of code is not a new one, the introduction of object-oriented programming to first year students has led to an explosion in the scale of reusable code that is available to students in code libraries. For instance, the standard C library contains approximately 200 functions that can be used by

programmers [Plauger1992]. We can compare this to the standard Java class libraries that contain almost 1500 classes, with each class containing tens to hundreds of methods [Gosling1999]. Because of the size of the class libraries that have to be dealt with, effective browsing techniques have become important.

The searching techniques described in section 2.3.3 are useful where specific information is known about the class desired, but there are many situations where the student may just have a vague feeling for the type of class that they require. Browsing a class library lets them gain a broad overview of the classes available and where to find them. It is important that the browser does not overwhelm the student by presenting too many classes at once. To achieve this, the browser must select a subset of the classes to display and it must display brief yet pertinent information about each class. Section 2.3.5.1 discusses this in more depth. Because only a subset of all the classes is displayed, it is also important that the student can navigate amongst the subset of classes, quickly moving through the classes in order to find the desired one. Section 2.3.5.2 talks about this navigation in detail. Sections 2.3.5.3 and 0 provide specific examples of browsing tools.

2.3.5.1 Visualising relationships

There are many options for how classes in a browser will be displayed. In the simplest case, the classes could be displayed alphabetically by their names. More commonly, browsers will use a graphical notation such as UML [Fowler1997] that shows various forms of relationships between the classes being browsed. Each node of the display is a class and these nodes may be augmented by colour or pattern to indicate other features of the classes. When a diagramming technique is being used, browsers may allow the author of the classes (or perhaps the user of the browser) to manually layout classes in a diagram. Another possibility is to automatically layout classes according to some algorithm [Seemann1997].

A common technique for browsers is to display the inheritance relationships between classes in a tree form with collapsible branches to allow the user to view only the parts of the tree in which they are interested. This form of browsing

becomes problematic in languages with multiple inheritance because the relationships cannot necessarily be described in a tree form.

Whilst inheritance relationships are the most common relationship used for browsing it is interesting to consider what other relationships may be used. Broadly, they can be broken into two categories. Technical relationships such as inheritance and dependencies are normally part of the metadata of the system, either retrievable directly from a languages' reflection interface or else easily derivable from the source code. These relationships are often able to be browsed in systems because they can be calculated automatically.

The other broad area is semantic relationships such as functional similarity. For example, it may be useful to view only those classes that serve a similar purpose to a list class such as a stack or a queue. Another possibility is the desire to browse classes based on whether two classes are often used together. For instance, it may be useful in Java to see the relationship between the `HTMLDocument` class and the `URL` class as these classes are often used together. Semantic relationships are not commonly supported because the information required to make them work must be supplied manually and is difficult to keep up to date. The automatic inference of these semantic relationships is an interesting area for future research.

The success of all these visualisation forms depends on the complexity of the classes being browsed. If there are too many classes, the student may not be able to form a mental picture of the classes' functionality [Pintado1990]. Some form of filtering may be required to restrict the display in this case.

2.3.5.2 Navigating relationships

The relationships between classes form an obvious basis for navigation because the relation defines a commonality between two classes. If we are looking at one class which does not quite fit our needs we can navigate to an ancestor in the inheritance hierarchy or perhaps another class with a similar usage pattern as this gives us a good chance of finding the class we desire.

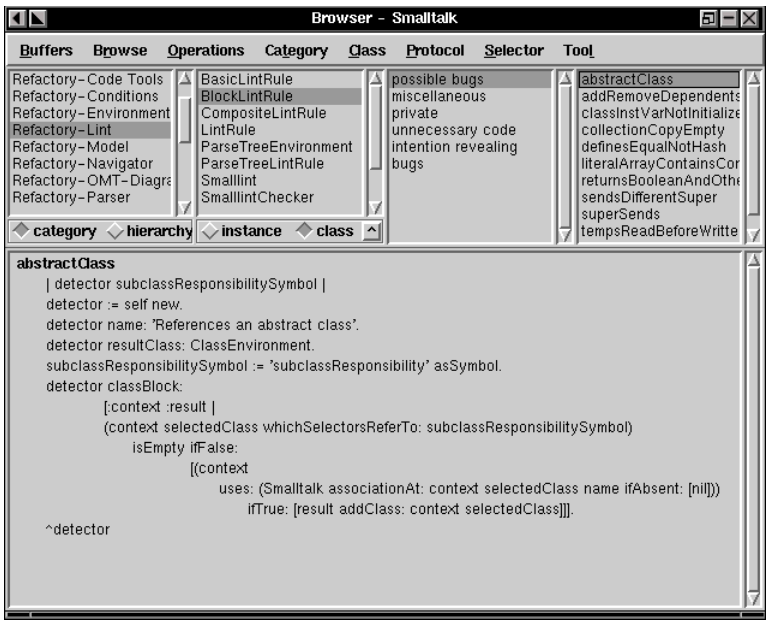

Figure 3 - A Smalltalk Browser.

It has been noted [Cook1992] that whilst the inheritance hierarchy is a good navigational relationship for developers who have authored classes, it is not necessarily the best for students looking for classes. Students who are looking for a class to use should start with the most specific (the leaves of the inheritance tree). The problems with navigation are similar to those with visualisation in that some of the semantic relationships that would form an excellent basis for navigation are difficult to enter and maintain.

2.3.5.3 Smalltalk

The Smalltalk environment is interesting because it was the first object-oriented programming environment and despite many different implementations over the years, current Smalltalk implementations retain much of the same look-and-feel as the earliest versions. Central to this look-and-feel is the importance of a class browser within the environment.

Smalltalk class browsers are structured as a row of scrollable list panes, each displaying a different level of granularity. The left most list displays general categories, the centre left list displays class names that belong to the chosen general

category, the centre right list displays categories of methods within a class (such as initialisation, private) and finally the rightmost list displays the actual method names of a class. When a method is selected in the rightmost pane, its source is displayed in the bottom pane (see Figure 3). An alternative approach that is used by some Smalltalk systems is to replace the two leftmost panes with a class hierarchy displayed as a tree structure (Smalltalk only allows inheritance from one parent class and all classes must have Object as an ancestor so the display of the classes in a tree structure is trivial).

Browsing in Smalltalk is like looking at an extremely large tree structure with a view at a number of fixed heights. Each view can navigate amongst its siblings but cannot move up or down the tree and changing the view at the top heights automatically moves the lower views across into the same sub-tree.

There are a few pitfalls with the Smalltalk form of browsing. Firstly, it requires accurate categorisation of both classes and methods in order for the programmer to be able to find source they want. Not only must programmers take the time to categorise their classes, differences in interpretation between programmers could lead to confusion as to the category to which a class belongs.

In the Smalltalk systems that display the class hierarchy in the leftmost pane a new problem is introduced. Rather than navigating a single tree with a uniform mechanism, introducing a class hierarchy in the leftmost pane means that the hierarchy tree must be navigated *within* the pane (options for this are displaying the whole tree or allowing sub-trees to be expanded and collapsed) and then another tree is navigated between the panes. This dual interface mechanism is cumbersome and confusing.

Another pitfall is that because there are a fixed number of heights, the tree of classes and methods can become extremely broad. This problem is overcome in modern object-oriented languages by allowing nested namespaces (for instance packages in Java allow the creation of a hierarchy of namespaces). In Smalltalk

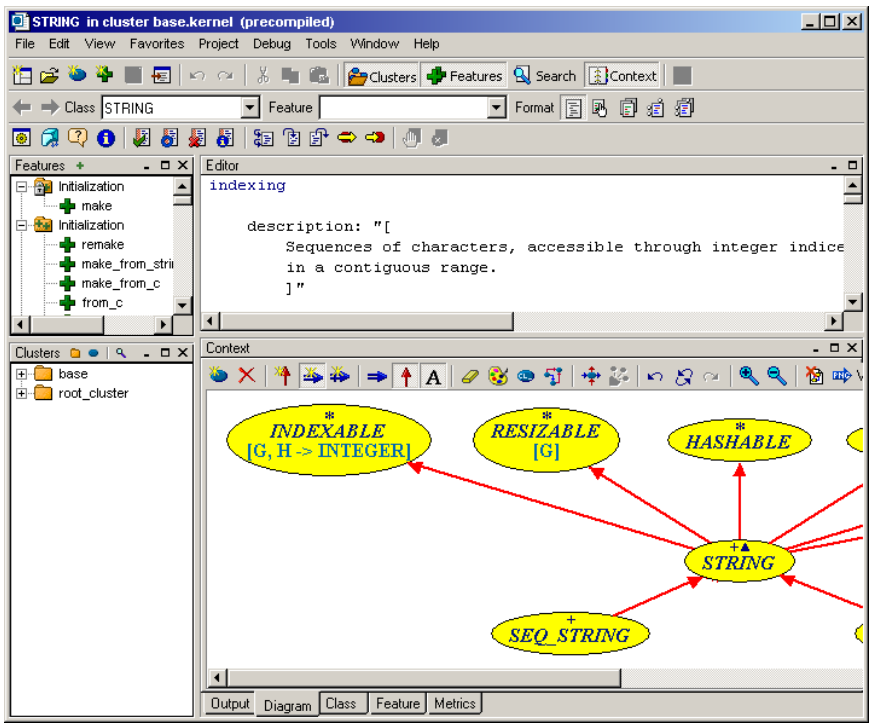

Figure 4 – A "Development Window" in EiffelStudio targeted on the STRING class (reproduced with permission from [Meyer2001 page 20]).

however, the lack of nested namespaces can create a broad tree that is overwhelming for students.

2.3.5.4 EiffelStudio

EiffelStudio [Meyer2001] is a development environment for the object-oriented language Eiffel. Browsing is considered particularly important in the EiffelStudio environment because "of the speed at which you can construct sophisticated class structures, making use of inheritance, genericity, the client relation and information hiding…" [Meyer2001 page 17]. Rather than launching a separate browser, the EiffelStudio environment is always in a browse mode, no matter whether editing or debugging. Each "Development Window" (many different windows can be opened at once, each displaying something different) of the environment targets either a class, feature, cluster or runtime object in the system. For instance, when a class is targeted, its source is displayed in the editor pane whilst other information about it is displayed in the context pane. This other information can be simple information

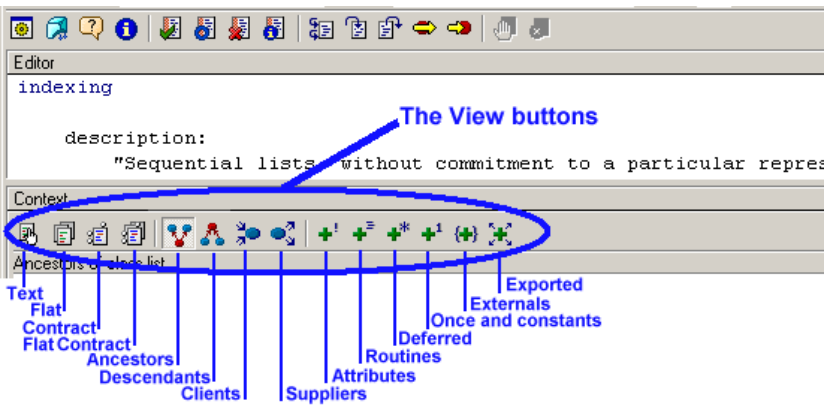

Figure 5 – The "class" view (reproduced with permission from [Meyer2001 page 31]).

such as its location on disk or more complicated information such as a diagram showing the classes' inheritance hierarchy (see Figure 4).

The user can browse to another class through numerous methods

- They can type the name of a class into a search dialog to directly target that class;
- Whenever the name of a class appears in the editor window it acts as a hyperlink which can be clicked on to target that class;
- They can go forward and backward through the classes that have been targeted by using back and forward navigation buttons; and
- Classes can be added to a "favourites" menu which allows them to be targeted quickly.

One interesting feature of EiffelStudio is the "class" view (see Figure 5) that displays information about a class with a variety of filters to allow only certain types of features to be displayed. Allowing alternative views of classes is a valuable way of managing the complexity of object-oriented classes.

Like Smalltalk, Eiffel has a flat namespace for classes. Unlike Smalltalk however, Eiffel has a nested mechanism to help organise classes into groups, thereby overcoming the broad trees that can overwhelm programmers within a Smalltalk environment. The grouping mechanism is called "clusters" and it relates directly to the directory structure used to store the Eiffel files on disk, though the on-disk structure is hidden from the programmer when using the EiffelStudio environment.

When viewing classes in "cluster" view, EiffelStudio allows the programmer to manipulate a diagram displaying relationships between classes (see Figure 6). Classes can be hidden on the diagram if they are deemed unimportant, reducing on screen clutter. All changes that are made to a class are automatically reflected in the "cluster" view diagram. For instance, if a class is added as an ancestor of another by editing its source, the diagram automatically displays this relationship. Similarly, changes can be made to the diagram that are automatically reflected in the classes' source code.

The "cluster" view not only shows the cluster that is currently targeted, it also shows those clusters that are children of the current cluster. This allows classes to be dragged and dropped between clusters, quickly and easily allowing the organisational structure of a project to be changed.

2.3.6 Build programs

Most languages and programming development environments have a build phase where the source code of the project is compiled into an executable program. Students must become familiar with the process of building a project, learning both how to specify the details of the project to be built and how to invoke the build. A program development environment can simplify configuring the details of a project (providing a user interface to various compilation options for instance) as well as simplifying the process of launching the compiler. If a program development environment is not being used then a tool such as Make or ANT [Ant2002] can be used as a way of specifying the details of the project to be built.

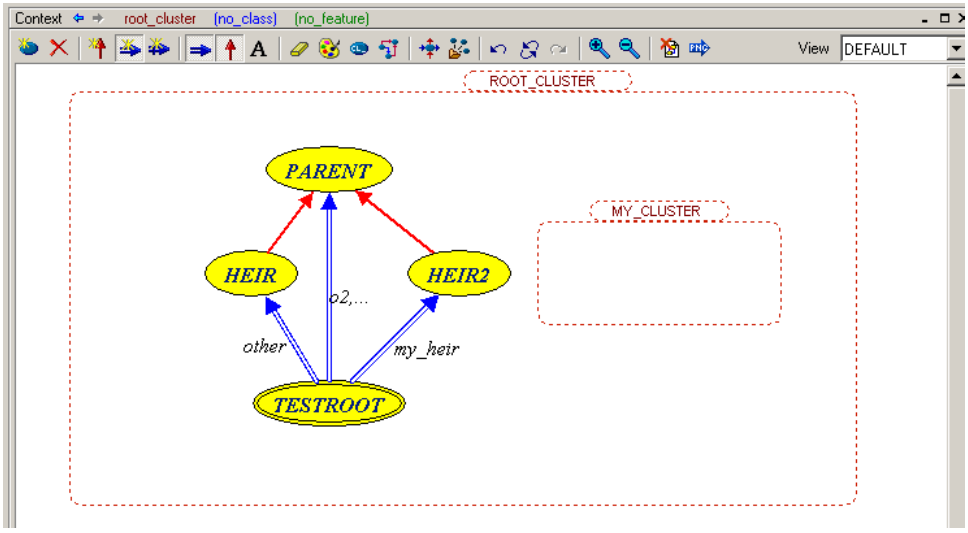

Figure 6 – The "cluster" view in EiffelStudio showing the empty MY_CLUSTER that has just been created as a child of the ROOT_CLUSTER (reproduced with permission from [Meyer2001 page 21]).

Not all systems have an explicit build procedure. Some examples are interpreted languages, and some modern systems such as Visual Age for Java where the build process is a continuous one, compiling fragments of code as soon as they are entered in the editor [Nilsson2000].

2.3.7 Implement and execute test cases

The development of a project does not end once code is written. After coding, it is important that a student test the project to make sure it behaves as intended. The most simplistic tool support for testing is one where the environment provides hooks to launch the application and capture results. This type of support is very coarse grained in that the application must be tested as a whole. When using an object-oriented language however, the natural units that the students is dealing with are classes and objects. A more advanced tool should allow a finer grained approach by supporting testing of these abstractions. But what level is appropriate for introductory students? Do current tools go far enough in supporting testing for introductory students? We will look at two possible levels that testing support could be added at, the class level and the object level.

2.3.7.1 Class level

Testing at a class level is often done by requiring students to implement a main method in every class that performs a test on the class. Each class can then be run as an application in order to see the results. Of course, there are problems with this approach if a class has more than one test. Then, students are forced to comment out sections of the main that perform one test in order to isolate the test they want to run. In the end, tests become outdated while they remain commented (and uncompiled) in the source. A better approach is to use some sort of testing framework to organise the tests in each class.

Some development environments have now integrated testing frameworks such as the JUnit [JUnit2002] test framework. The JUnit testing framework defines a set of interfaces that define how tests behave (`Test`, `TestListener`) and a set of implementations that can be extended easily to use their testing functionality (`Assert`, `TestCase`, `SwingRunner`).

A JUnit test is a standard Java class that inherits from `junit.framework.TestCase`. Within it are methods with names such as `testAddition()` or `testSort()`. Methods can contain assertion statements which assert a particular condition as part of the test. JUnit provides a GUI or text based "runner" that executes test cases and displays which of the assertions in the test methods failed.

The integration of JUnit into development environments is still quite primitive. Most provide no facilities other than the ability to launch the test "runner" and a rudimentary designation of classes in the environment as "test" classes. Some now provide the ability to automatically generate test classes with stubs based on existing classes.

More details of JUnit are provided in section 4.4.2.

2.3.7.2 Object level

Testing at an object level should allow the student to interact with objects in the system and test individual methods on these objects. There are very few environments that support this fine grained testing of object oriented code. Some systems that do allow a limited form of object oriented interaction are discussed as part of the "Running and debugging" task in section 2.3.8 . None of these systems integrates the object interaction with any other testing facilities they may have. We have identified this area as one that lacks appropriate tools and this has motivated our work on developing an object testing facility for introductory students. This work is discussed in chapters 4 and 5.

2.3.8 Run and debug applications

Writing and executing tests is an activity that is often performed on completion of the coding phase, either after the whole application is developed or preferably as individual classes are completed. However, there is another activity that is akin to testing, that is performed whilst the code is being written. This form of interaction is performed at runtime and involves the inspection of the state of objects in the system and the examination of the behaviour of running code. Most environments support this activity through the use of a symbolic debugger. The symbolic debugger allows breakpoints to be set on code that interrupts the execution and allows the runtime state to be inspected. Symbolic debuggers behave in much the same way they did when students were developing with procedural languages. Few of them have the ability to visualise the structure of the objects in the system or interact with these objects.

There are some environments that allow interaction with objects in a direct and more object-oriented manner. Whilst most are not specifically designed for students, they have interesting ideas for ways of dealing with object interaction. A few of these environments are discussed in the following sections.

2.3.8.1 Smalltalk systems

With Smalltalk being one of the first object-oriented languages with a development environment, it is natural that one of the first systems involving direct interaction with object instances was developed within a Smalltalk environment. Portia is an enhanced Smalltalk environment that is *instance-centred* in that it "provides facilities for working directly with objects to debug, understand and create applications" [Gold1991 page 62]. Portia enhances the Smalltalk environment by adding the ability to drag and drop instances of objects into the existing Smalltalk debugger, class hierarchy browser and inspector. It also adds an object repository whose purpose is to collect and hold object instances. Objects can be dragged from this repository into any of the other tools and vice-versa. Utilising standard Smalltalk techniques it is possible to simulate method calls and see how objects behave. Portia mainly adds ways to manage the complexity of dealing with thousands of objects and easy ways of locating and dealing with the objects of interest in a system.

The authors of Portia raise a very salient point regarding the usefulness of direct object interaction. "Existing objects can furnish a wealth of information about their behaviours" [Gold1991 page 63]. In a perfect world, all classes would be adequately documented and so there would be no need for this interaction. However, in a world where sometimes documentation goes astray or falls out of line with the actual implementation, the ability to see how an object behaves by directly interacting with it is an extremely useful technique.

2.3.8.2 Self Environments

Seity [Chang1995] is an experimental user-interface for the protoype-based object-oriented language Self. The premise of Seity is to move away from *view-focused* environments and move to an *object-focused* model. The authors of Seity define a view-focused environment as one in "which objects are examined and manipulated through intermediaries, each of which permit a certain view of the objects" [Chang1995 page 2]. For instance, although the Smalltalk object inspector displays a particular object at any one time, over a period of time the tool's window could be

used to display a variety of different objects. Whilst recognising that with frequent use, a programmer may begin to regard the object inspector tool as the object, the claim is that this abstracts and distances the objects. Seity's object-focused model says that "the on-screen representation of the object is considered to represent the object itself, not merely a singular tool through which the object shows itself" [Chang1995 page 3]. This helps to reinforce to the student the notion of what an object is. Unfortunately, the Seity environment is quite limited in other functionality that is needed for introductory students and prototype languages such as Self have not become popular. Hence the Seity system is not still in active development.

2.3.8.3 Object Class Browser

One of the few environments to support object interaction in Java is the Object Class Browser (OCB) that was written initially in the context of the PJama persistent Java environment [Kirby1997]. A persistent object environment can potentially contain many thousands of objects and may require multiple different techniques to help manage and discover these objects. The OCB provides one technique which is to visualise each object in the system as a window displaying the object's fields and which allows navigation to other objects in the system through references from these fields. OCB is implemented entirely in Java and uses the Java reflection mechanisms to inspect the objects in the system. OCB handles potential confusion over object identity by only allowing one window to exist for each object instance, and if a window already exists for an object that is navigated to, then this existing window is brought to the front. In this respect it is similar to the Seity environment discussed above and similar caveats apply.

2.3.8.4 DrJava

The DrJava environment is a lightweight programming environment for Java with a pedagogical focus [Allen2001]. It has a deliberately simple read-eval-print loop (REPL) interface that aims to minimise the intimidation students feel when faced with writing code. The interface consists of two windows; an interaction window

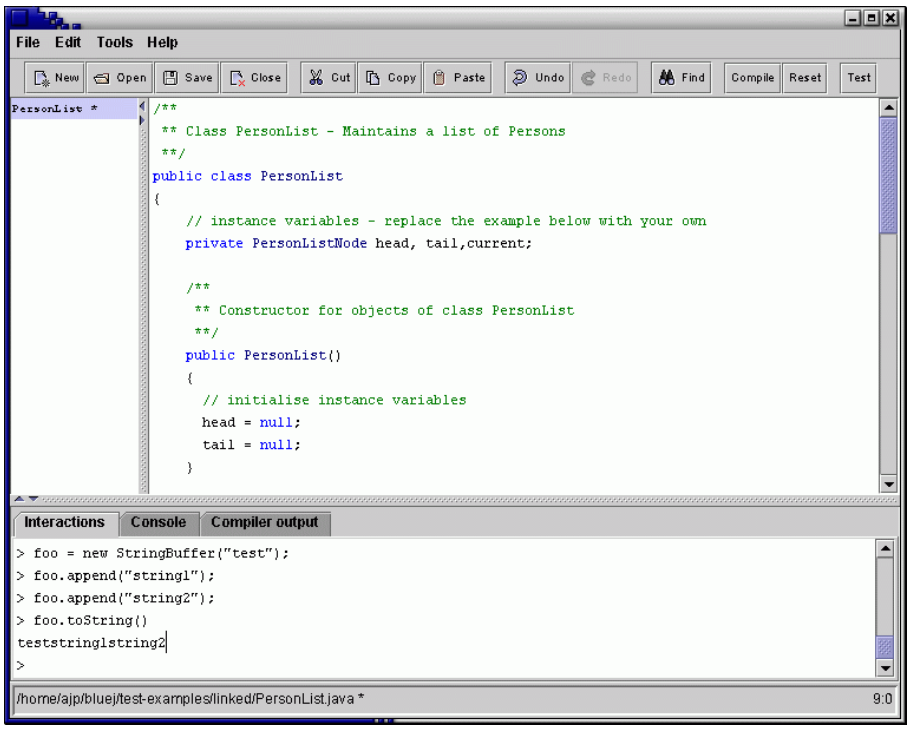

Figure 7 – DrJava showing object interaction being performed in the lower panel.

where Java expressions can be entered and the results are displayed, and a definition window where Java source for class definitions can be entered (see Figure 7).

The REPL that is the main interactive interface for DrJava is a carryover from an earlier project called DrScheme, which provided similar features for the Scheme language. Whilst the REPL is a natural fit for a functional language such as Scheme, in an object-oriented language such as Java the benefits of using it as the sole interactive interface are not so clear.

We agree with the DrJava authors that a REPL provides some useful functions for students:

- the students can write simple expressions to experiment with the language and see how it behaves;

- methods can be tested directly by executing them from the interaction window. The need for a main method with test code as an entry point to each class is obviated; and

- the students can use the interactive environment to explore the standard Java APIs by instantiating standard objects and seeing the results of method calls on these objects.

However, the disadvantages are that a REPL hides from the student the important concepts of object-oriented programming, namely objects and classes! Unlike some of the other object interaction systems we have discussed, the DrJava interface does not emphasise the distinction between objects and classes, does not reinforce the notion of object identity and relies on implementation of a toString() method for each class in order to examine object state.

2.3.9 Refactor code

Refactoring is the "process of changing a software system in such as way that it does not alter the external behaviour of the code yet improves the internal structure" [Fowler1999].

Tool support for refactoring is the next stage in the evolution of refactoring as a software methodology. Automatic tool support makes the time-cost of refactoring negligible and makes refactoring less of a separate activity and more of an activity that is performed hand in hand with writing code. Roberts and Brant, authors of the Smalltalk Refactoring Browser, have developed some criteria, both technical and practical, which they believe are most important for tools implementing refactoring [Roberts1999b]. The technical criteria are:

- the development of cross reference information for the project being refactored;

- the ability to manipulate parse trees of the language being refactored; and

- the ability to ensure that refactorings are accurate and reasonably preserve the behaviour of programs.

The practical criteria are that the tools are:

- fast enough that they do not impair the developers work flow;
- support the concept of undo; and
- are well integrated into the development environment.

In the following sections we describe two examples of tools that support refactoring.

2.3.9.1 Smalltalk Refactoring Browser

The first attempt at a tool that could automate the steps of refactoring was the Smalltalk Refactoring Browser [Roberts1997]. The refactoring functionality was initially integrated into the standard Smalltalk browser although in later versions it was implemented as part of a completely new Smalltalk browser. The refactorings supported are adding, removing and renaming methods, variables and classes. It is also possible to perform a pull up and push down on methods and fields, i.e. moving them to an ancestor in the inheritance hierarchy. Some of the more complicated refactorings it can perform are adding parameters to a method and extracting code as a method.

There were three criteria used in building the refactoring browser. One was that it worked with standard tools and this was achieved by integrating it with the standard Smalltalk browser. Second was that it had to be fast, since Smalltalk programmers are used to immediate feedback with their system. Thirdly, completely automatic organisation was to be avoided due to the importance of naming in the Smalltalk language. In cases where a refactoring requires the creation of a variable or a method, the user is prompted to enter the name rather than the system attempting to calculate it automatically.

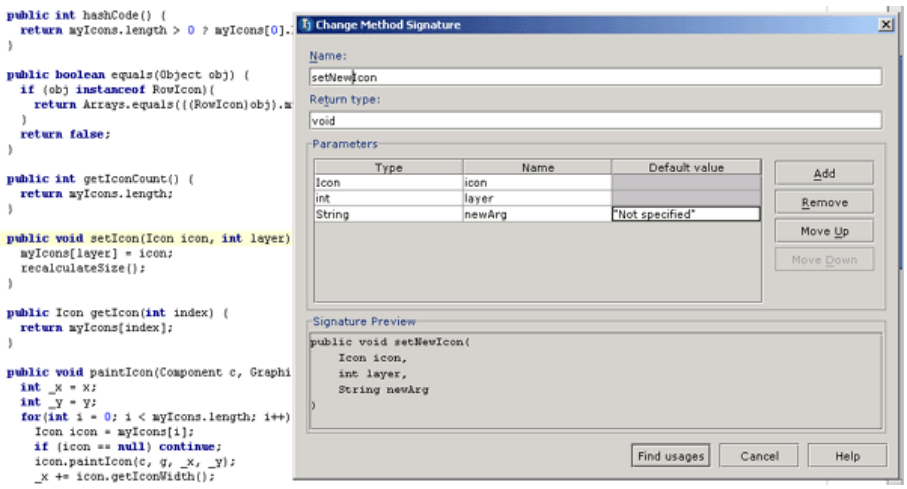

Figure 8 - Performing a refactoring with IntelliJ IDEA.

2.3.9.2 IntelliJ IDEA

The IDEA IDE from IntelliJ [IntelliJ2002] is an advanced IDE for the Java programming language. It is one of the first Java IDEs to support advanced refactorings such as extracting code as a method and changing method signatures (see Figure 8). It also supports class, method and field renaming and moving. When evaluated against Roberts and Brant's criteria for refactoring tools [Roberts1999b], IntelliJ passes with flying colours.

Progress in tool support for refactoring has been very rapid with multiple refactoring IDEs for Java appearing recently. We envisage that the number of different refactorings supported by tools and the robustness of their implementations will also improve rapidly over the next few years.

One aspect that has not been addressed is the application of refactoring tools in introductory teaching. Clearly some of the refactorings use advanced concepts and are only needed in large, long lived projects. In chapter 3 we address a number of issues related to refactoring. These include whether there are any refactorings that are suitable for first year students and whether refactoring should be presented to students with a tool that guides them to the correct refactorings.

2.3.10 Integrate external resources

When class libraries are obtained from external sources, be it libraries from external vendors or class libraries from other students, students need to make their development environments aware of the library to enable it to be used. Students may merely have to place the library in a certain location for it to be automatically recognised or may have to perform some manual configuration steps to integrate it into their system. Some other functionality may allow code to be reused from a centralised code repository without any intervention by the student. This is ideal for courses where the instructor wants to make new class libraries available to students as the weeks progress.

Tied in with the issue of integrating class libraries is the issue of how applications are eventually distributed. If the class library is distributed as a single unit, does this unit get integrated into the resulting application or do the users (perhaps through a special installation program) need to perform integration steps to add the class library to their systems before the application will run? Development environments with support for this can greatly ease handling of students' assignment submission.

It is normally quite straightforward to integrate a class library into a development environment. In Borland's Delphi, the student selects "Install Package…" from the menu and then selects the package that they wish to integrate. Another example is Sun's Java Development Kit (JDK) in which code libraries can be added to the system namespace by placing the compiled files into a system "extension" directory. Alternatively, the class library can be specified explicitly each time the JDK is used.

2.4 Summary and motivation

In this chapter we have looked at a variety of approaches for teaching the knowledge units that make up the software engineering body of knowledge. In particular, for the software product engineering area we have examined the role that software tools can play in facilitating the teaching of introductory students. Additional evaluation of some of these tools can be found in [Kölling1999].

To aid our evaluation, we have identified ten tasks that cover most of the activities that an introductory student will perform. For each of these tasks we have discussed the use of tools and highlighted those that provide particular pedagogical value.

In the area of design, we have concluded that tool support for drawing system designs is adequately provided by professional UML tools for advanced students, and simple drawing tools for introductory students. We have also concluded that graphic user interface builders are suitable for students to construct user interfaces.

In the area of coding, current tool support for building programs, searching documentation, and entering and editing programs is more than adequate. There are many tools for browsing class libraries available but not many that integrate seamlessly into current program development environments. It is surely only a matter of time before all mainstream development environments include class browsing facilities.

In the area of operations, the integration of external resources is supported satisfactorily by most development environments.

In the area of maintenance, tool support for refactoring has advanced rapidly. However, we do not believe that refactoring tools are designed with a pedagogical focus. In chapter 3 we look at refactoring in an educational context and develop a design for a refactoring tool in an existing program development environment.

In the area of testing, we find a scarcity of tool support that is accessible to introductory students. Most tool support for testing is based on techniques that were used in the days of procedural programming languages. Despite some tools that allow object-oriented interaction with programs, these tools are not always suitable for students and none has combined this object interaction with support for repeatable testing. In chapters 4 and 5 we look at the development of an object oriented test support facility in a program development environment.

3 Chapter 3

REFACTORING

The previous chapter has identified areas of product software engineering that were lacking in suitable tool support for introductory students. This chapter looks at one of these areas, refactoring, and discusses the design of a tool for refactoring suitable for an introductory integrated development environment.

3.1 Introduction

As mentioned in the previous chapter, refactoring is the "process of changing a software system in such as way that it does not alter the external behaviour of the code yet improves the internal structure" [Fowler1999]. Refactoring is one aspect of the trend in software engineering that recognises the fact that it is extremely difficult to design object-oriented programs correctly. This is a particularly serious issue for introductory students who do not have any personal design experience to fall back on to help guide their designs. Refactoring also recognises that the development of software is an incremental, continuous, evolving process – software will change as its purpose and requirements change. Refactoring, along with some other new development techniques such as pair programming and unit testing, form part of the new development methodology called extreme programming (XP), which better handles the process of continual change in software development [Jeffries2000] [Beck1997].

There are other schools of thought on how to improve our ability to design object-oriented programs. Some of these attempt to identify common design patterns that are known to be good solutions [Gamma1995], or alternatively, identify common design mistakes in order to prevent them occurring [Brown1998].

Refactoring accepts the reality that it is unlikely that a perfect (or even good) design will be realised before coding starts, and therefore concentrates on techniques that

will safely allow evolutionary changes to be made to the internal structure of the program. Refactoring also faces the reality that, even with an excellent design, the needs and functional requirements of a program will change over time. Unless programmers implementing the new functionality have a full understanding of the original design, their changes may tend to decay the structure of the design.

Fowler [Fowler1999] identifies the following four reasons why programmers should refactor:

- Improves the design
  Programs, especially those that have had a long life span, tend to accumulate redundant code, obscuring the original design. Refactoring can help move any code that is in the wrong place to the right place, can help to eliminate redundant and duplicated code, and thereby restore structure to the program;

- Makes software easier to understand
  There are two phases in the life of code, the phase where the code is written and then a phase of maintenance. The second phase may occur months after the code is initially developed and may involve a different programmer than the initial developer. Refactoring code to make the design clearer will help the understanding process of programmers later down the track;

- Helps you find bugs
  The clearer the design and structure of a program the easier it is to write robust code and the easier it is to spot logic flaws or mistaken assumptions (because the clear structure emphasises the assumptions made). Thus, refactoring can reduce the number of bugs in code; and

- Helps you program faster
  Despite the extra time taken to perform refactorings, the maintenance of a clear design and structure in the program can lead to continued rapid development (rather than not refactoring, which can start the coding process

quickly but which tends to complicate coding as the initial design starts to decay).

Of course, refactoring is not necessarily going to be successful in every situation. There has been very little research into the pitfalls of refactoring. Anecdotally, it has been reported that refactoring which involves modification to database schema as well as code may be problematic. Similarly, code with published interfaces (such as when the Java collection classes were introduced into version 1.2) is difficult to refactor because many refactoring steps involve changing the interface. In some cases, the code may be beyond repair and a complete rewrite may be more effective than refactoring.

3.2 Why refactor in first year?

We have discussed refactoring and its importance in the maintenance and evolution of code. What then is its usefulness in introductory education given that most introductory assignments are small-scale projects or projects where there is no maintenance component? Is there any value to teaching refactoring techniques to first year students?

We believe that refactoring is an appropriate skill to teach in first year because it helps students reach the goal of developing well-structured programs. Whilst students may not need to perform real maintenance on any of their projects, we want them to get into the habit of re-evaluating and restructuring the code they develop. Refactoring is an appropriate skill not just because it will be a skill that is useful in larger projects, but because it supports one of the goals of teaching object-oriented programming, that is, finishing with a well designed and structured program.

Traditional teaching has viewed software along the lines of the waterfall development model – a staged process that moves from design to implementation to testing, and where each stage is unchanged once completed. The growing awareness that the waterfall model is unrealistic in the real world, has led to a shift

in the model now taught to students. Students must recognise that software is an ever changing, ever growing artefact that requires constant maintenance as it adapts and meets changing needs. Preparing students for this requires that they be taught different skills, including the ability to evaluate their designs and refactor their code.

3.3 What refactorings are appropriate?

A list of refactorings has been collected by Martin Fowler on his refactoring.com website [Fowler2002]. The list includes all those in his book [Fowler1999] as well as those contributed by people around the world. Currently there are almost 100 refactorings that have been catalogued. Only a small proportion of these refactorings are useful for a first year student. An even smaller proportion can be aided by a refactoring tool. We wish to identify those refactorings that would be useful in an introductory refactoring tool. The following criteria will be used to consider which refactorings to support:

- Can be automated

  Some refactorings require a complex understanding of the source code or require an understanding of the way in which code is used that is outside the scope of what could be inferred by machine analysis. These refactorings are difficult to automate. Often, these complex refactorings can be performed manually as a sequence of smaller basic automated refactorings; and

- Occurs in student sized projects

  Some refactorings are not useful to consider because they would never occur in the types of projects that students will work with in introductory courses. An example of this would be the "Duplicate Observed Data" [1] refactoring which involves constructing observers on GUI controls; and

[1] If there is domain data available only in a GUI control and domain methods need access, copy the data to a domain object and set up an observer to synchronize the two pieces of data.

- Has no intrinsic value in being performed by hand
  Performing a refactoring by hand may itself have some value. For instance the "Replace Conditional With Polymorphism" [1] refactoring has pedagogical value in requiring a student to perform it manually. After applying the refactoring, the student will have an improved understanding of how polymorphism works and hopefully not write the incorrect code the next time around; and

- Will be used enough to warrant cluttering an interface
  Each refactoring that is included in our in introductory tool will in some way complicate the interface that is presented to the user for selecting refactorings. Because we are designing an introductory tool, clarity in a user interface is extremely important and hence trading off the usefulness of a refactoring with the additional clutter it brings to the interface is another criteria.

We will not attempt to evaluate all refactorings against these criteria. Rather, we identify various categories of refactorings and list some key examples of these types of refactorings. We then evaluate these general refactoring categories against our criteria.

Of course, it is not possible to definitively identify each, and only those, refactorings that are appropriate for an introductory refactoring tool. The suitability of many refactorings will depend on the structure of the introductory course being taken, the type of material that is covered and the extent to which the refactoring tool may be used in latter courses. We will identify some refactorings as "borderline" candidates. These refactorings could be suitable for an introductory tool but we have decided not to include them in the design of our refactoring tool.

[1] If a conditional statement makes decisions based on the type of an object it should be replaced with a polymorphic method call.

3.3.1 Changes local to a method fragment

The following refactorings all deal with improving fragments of code within a method body. Some require the construction of a new method, but these new methods can be private methods and will not affect the public interface of the class. Whilst it is possible to automate some of these, the localised nature of the changes means that they can be made quite quickly by hand and tool support is not required. There is a large group that deal with conditional statements. Some examples are:

- Consolidate Conditional Expression
  Replace a group of conditionals that all return the same value with a single method call to check all the conditionals;

- Consolidate Duplicate Conditional Fragments
  If the same code is contained in multiple conditional fragments it should be moved to outside the conditional statement;

- Decompose Conditional
  Simplify a complicated conditional statement by adding query methods for the complicated expressions; and

- Replace Nested Conditional with Guard Clauses
  Clarify the expected path through a conditional statement by using guard clauses.

Another group deals with the use of local variables and their scope:

- Reduce Scope of Variable
  Reduce the scope of a variable because it is only used in a small fragment of a method body; and

- Split Temporary Variable
  The use of a temporary variable for two unrelated tasks in one method should be replaced with the use of two different temporary variables.

Tool support could be added to support these two refactorings by performing a simple analysis of the usage of a variable within a method. For instance, when the editor cursor is placed within the definition of a local variable, the region from the first initialization of the variable to the last usage in the method could be discretely highlighted (perhaps by a small change of colour to the background of the editor in the region). This would clearly show the scope where the variable is actually being used and might indicate that a lesser scope could be used. Similarly, if the colour of the background was changed slightly upon reaching the second assignment to a local variable, multiple use of a temporary variable could be shown.

In summary, the localised nature of the changes of these refactorings means that in most cases performing the changes by hand is quicker and safer than constructing an automated refactoring tool. For this reason, these refactorings are not considered for our introductory tool.

3.3.2 Changes to a method signature

Changing the signature of a method is one of the key refactoring operations that can be aided with a refactoring tool. The difficulty in performing these refactorings by hand is that all places in the source code that refer to the method must be identified and changed. An automated refactoring tool collates all the references to a changed method and allows them to be updated automatically. The basic method signature refactorings are:

- Add/Remove Parameter
  Add or remove a parameter to a method call. Removing a parameter is a simple operation but adding a parameter requires providing a default parameter value; and

- Rename Field / Method
  Change the name of a field or method.

These refactorings meet all our criteria, and because they are also crucial to many other refactorings, it is essential that they are presented in our introductory refactoring tool.

3.3.3 Changes to a class structure

A large group of refactorings deal with changes to the structure of classes. Some involve the splitting or merging of entire classes. Others involve moving methods and fields to a different class, or up and down between classes in an inheritance hierarchy. We will consider each of these groups in turn.

3.3.3.1 Move operations

Move operations are easy to automate as long as one can analyse a system for references to a class, method or field [Power2000] [Dewhurst1987a]. Then all that needs to be done is to correct those references so that they now refer to the new class, method or field location. These operations are good candidates for an automated tool, not only because it is clear how to automate them, but also because there is no benefit in making a student perform the laborious task of making many simple typing corrections. By supporting these refactorings in a tool, students can concentrate on the high-level conceptual task of making structural changes without being distracted by the low-level mechanics of performing the task. Additionally, by lowering the barriers to performing these tasks it is much more likely that students will do them.

In the case of moving a non-static method or field, the reference will not always be correctable because a reference to an object through which the method or field will be accessed may not be held at the reference point. In this case, the best an automated tool can do is collect these reference locations and present them to the programmer as source locations that need to be corrected. Even though performing the corrections is then a manual task, having a tool collect the reference locations is still a valuable time saver.

The move refactorings are:

- Move / Rename Class
  Move a class from one package to another, or rename a class; and

- Move a Field / Method
  Move a field or method from one class to another.

An appropriate user interface for enabling these refactorings is development environments that present a high-level class overview. For instance, if a development environment presents a UML class diagram of the system, drag and drop or popup menus can be used to move methods and fields between classes and perform class rename operations. This allows high level design work to be performed on the complete system without having to look at individual class' source. Of course, it is also important that the move refactorings are also available when editing a class' source, though the interface may not be so intuitive.

3.3.3.2 Extract operations

The extract refactorings are important to students because they deal with the types of design mistakes that beginners often make. Typically students write large methods and classes because that is the path of least resistance – it avoids constructing objects and making calls on those objects that some students find intimidating. There is also an element of laziness in that constructing a new class involves creating a new file, setting up constructors and other mundane overhead before it can be used.

The intimidation felt using multiple classes must be overcome through improving understanding, but the laziness can be overcome by making the construction of a new class a trivial operation. Most development environments already automate the construction of a standard empty class. The "Extract Class" refactoring encourages students to create new classes from existing classes when they feel that their class has become too large. Similarly, the "Extract Method" can be used to quickly split a

large method into multiple smaller methods. Through automation, these operations become quick and painless and students are encouraged to perform them.

The extract refactorings will also be useful during the normal growth of a piece of software. As functionality is added, methods and classes naturally grow. At the point where the class or method is becoming unmanageable, it can easily be split into a new class or method. In many ways, this use of extract refactorings is similar to the design technique of using CRC cards, where classes are assigned responsibilities until they gain too many, at which point they are split into two classes [Beck1989].

The basic extract refactorings are:

- Extract Class / Interface
  Create a new class containing some of the fields and methods from an existing class or interface;

- Extract Method / Split Method
  Turn a fragment of a method into a new method with a name that explains its purpose;

Based on these basic operations are the "Extract Subclass" and "Extract Superclass" refactorings. As with the "Move Field / Method" refactoring, some references to the fields and methods will not be able to be corrected due to there being no reference to the new object at the original reference location. These original locations must be highlighted for the user to fix manually.

As with the other inheritance structure refactorings discussed in the following section, "Extract Subclass" and "Extract Superclass" are borderline cases for consideration in our introductory refactoring tool. A more detailed explanation of the rationale for their inclusion or exclusion is contained in section 3.3.3.3.

3.3.3.3 Inheritance structure operations

One category of structural changes that can be made to classes involves changes to the class' inheritance hierarchy. This category deals with moving methods and fields between subclasses and superclasses. The operations are all modifications of the basic move refactorings discussed in section 3.3.3.1.

Some examples of the inheritance structure refactorings are:

- Pull Up Constructor Body
  The constructor code for two or more subclasses is similar so the functionality is moved into a superclass constructor;

- Pull Up Field / Method
  All subclasses of an object have a field or method in common so move it to the superclass;

- Push Down Field / Method
  A field or method is only used in some subclasses, so move the field or method down into those subclasses; and

- Collapse Hierarchy
  A superclass and subclass are not very different so merge them together into a single class.

The construction of automated tools to perform these complete refactorings is difficult. Additionally, there is trade-off between cluttering an interface with these quite complex refactorings, and constructing a tool that can cope with the demands of students in the latter stages of CS1. We have chosen to not include these inheritance structure refactorings in our design because we believe having a simpler, less cluttered user-interface surmounts the usefulness of having these refactorings present for the potentially few times that they will be needed. Furthermore, it is always possible for students who wish to perform these refactorings to perform them as a sequence of other more basic refactorings.

3.3.4 Changes to the design

The final class of refactorings deal with what are termed "bad smells" in a design. A "bad smell" is a particular design that works correctly, but could be improved by applying a design refactoring. Examples are:

- Encapsulate Collection
  Rather than returning a read/write collection, return a read-only collection and provide a method to add to the collection;

- Encapsulate Field
  Replace a public field with a private field and accessor methods;

- Encapsulate Downcast
  Replace a method that returns an object that needs to be downcast, into a method that returns a more specialised class and performs the downcast within the method;

- Hide Method
  A method is not used outside a class so make it private;

- Replace Error Code with Exception
  Replace numeric error codes with code that throws an exception; and

- Separate Query from Modifier
  A method that makes a query and also sets a value in an object is split into two separate functions.

Applying these refactorings is done using a combination of the simpler refactorings discussed in previous sections and through changing the source by hand. For instance, performing the "Hide Method" refactoring merely involves changing the method definition from public to private. This is such a simple operation that it does not require automation. Applying the "Separate Query from Modifier" refactoring involves performing an "Extract Method" on the query section of the method and then manually editing the resulting methods.

Rather than automating these refactorings, we see interesting work on tool support in this area as investigating techniques for detecting "bad smells". The RevJava tool analyses design smells according to 80 built-in design criteria including dead code signalling, design pattern checks and scoping/visibility checks [Florijn2002]. A tool that highlighted design mistakes for students as they develop code would be very interesting.

3.3.5 Summary

We have examined a catalogue of refactorings to determine a set of refactorings that we consider should be included in a refactoring tool in the context of introductory teaching. We have rejected some refactorings because they are difficult to automate. Many of these difficult refactorings are composed of other basic refactoring operations and hence can be still performed by the user as a manual sequence of basic refactorings.

Other refactorings we have discussed consist only of local changes to a method body. These refactorings can easily be performed entirely manually and do not require tool support.

Another category of refactorings make changes to the inheritance structure of classes. These have been discounted because not only are they hard to implement automatically, but we believe that on balance, the benefits of providing them are outweighed by the additional complexity they would bring to a tool's user interface.

The following is the set of refactorings that we believe are appropriate for consideration in the design of our introductory refactoring tool:

- Move Class
  Move a class from one package to another;

- Rename Class
  Change the name of a class;

- Extract Class
  Extract a subset of the methods and fields in a class into a new class, handling all the routine initialisation tasks for the new class such as building constructors;

- Extract Interface
  Extract a subset of the methods in an interface into a new interface;

- Extract Method
  Extract a block of code from a method into a new method, handling all the details of determining the set of parameters which need to be passed in, and constructing the method head and body;

- Rename Method
  Change the name of a method;

- Rename Field
  Change the name of a field;

- Change Method Parameters
  Add or remove one of the parameters to a method call;

- Move Field
  Move a field from one class to another;

- Move Method
  Move a method from one class to another;

These refactorings have been selected because we believe they operate at a level that is appropriate for first year students. That is, in a modern introductory software engineering course, these refactorings can be used to improve the basic aspects of design that will be discussed in the course, such as increasing cohesion and reducing coupling. Students benefit from having a tool that can support these refactorings because it removes the incidental complexity of performing the refactoring and lets

them concentrate on the higher level design task. We can provide a good level of tool support for students with these refactorings because much of the low level tasks can be automated.

As we will see in the next section, tools that support this set of refactoring functionality are already available. We propose some changes to the user interface of these tools to better support introductory students.

3.4 Current tool support for refactoring

It is not only important to decide which refactorings should be supported, but also how this can be done in an appropriate way. Since the goal of educational tool support for refactoring is to remove incidental complexity of the tasks to allow the student to concentrate on the concepts, it is important to ensure that the tool itself does not introduce a large degree of added complexity in itself.

Every software tool adds some degree of complexity to the user interface that must be learned and dealt with. The challenge is to design a tool whose external complexity is clearly lower than that of the tasks it seeks to automate.

The question of a degree of complexity cannot be judged in an absolute way. Complexity of a tool is to a large degree a question of experience and prior knowledge, so we have to examine the complexity of possible tool designs in the context of our specific targeted user group: first year students. The aim must be to design a tool that is simple enough to not add much overhead for an inexperienced programmer who has little familiarity with their programming environment.

We have already looked at some tool support for refactoring in the previous chapter. We took note in particular of the IDEA tool from IntelliJ [IntelliJ2002], which provides all of the refactorings we have identified as worthwhile for an introductory teaching environment. However, IDEA has some limitations that impinge upon its usefulness for first year students.

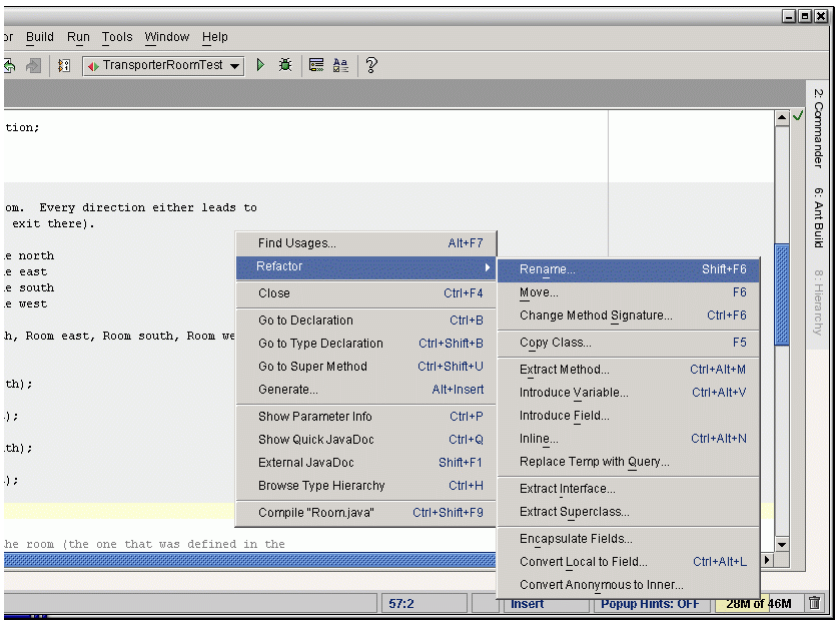

Figure 9 – The refactoring menu in IntelliJ IDEA.

We see the main problem with this tool is that it is designed for a professional computer programmer. The functionality available on the refactoring menu contains all refactorings that are implemented by the tool (some 15 refactorings in total). Adding to the problems with the large menu is that the menu is not context sensitive regarding the refactoring operations available at different points in the source. In Figure 9 we can see the popup menu presented when right clicking on a method's source in IDEA. All possible refactorings are listed, irrespective of whether they are actually available in the current situation. In fact, from the situation presented in Figure 9, only a handful of refactoring operations are applicable. The other menu items result in a dialog explaining why that particular refactoring cannot be performed.

Other tools provide a similar set of refactorings as IDEA, though with different user interfaces. The IBM Visual Age for Java development environment has better contextual popup menus that only shows refactoring operations available at the current location. We believe this contextual support is important in guiding introductory students to which refactorings are suitable.

	Operates on selected text in editor	Class operation	Field operation	Method operation
Move class		✔		
Rename class		✔		
Extract class		✔		
Extract interface		✔		
Extract method	✔			✔
Rename method				✔
Rename field			✔	
Change method parameters				✔
Move field			✔	
Move method				✔

Figure 10 – The context in which refactorings are appropriate.

The transmogrify tool [Transmogrify2001] is a library that provides support for analysing Java source and performing refactoring operations. It however has only a command line interface, though it does support integration with standard development environments through special hook classes. As the transmogrify tool is available for use under the GPL [GPL1991], it would be suitable as a back-end for an introductory refactoring tool assuming a suitable front-end interface could be developed. As refactoring is best performed during the development cycle rather than as a completely separate activity, we would like to integrate the transmogrify back-end into a development environment suitable for students. The next section will elaborate on our design for a refactoring tool using the BlueJ integrated development environment [Kölling2001a]. The BlueJ tool will not be presented in this chapter, as the only aspect of BlueJ that will be utilised for refactoring is the editor component. A more in depth look at the other features of BlueJ will be presented in chapters 4 and 5 when a unit testing extension to BlueJ is developed.

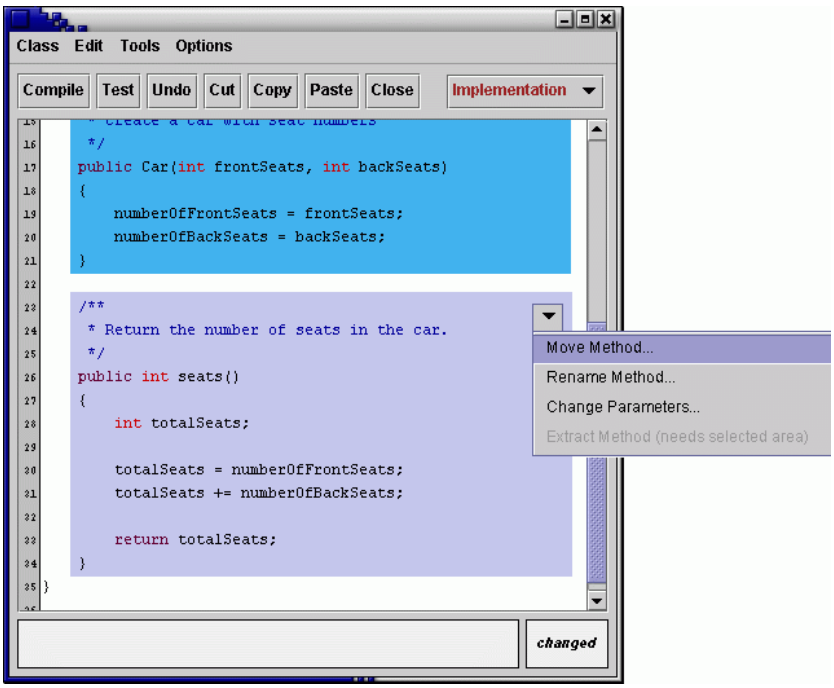

Figure 11 – The popup menu attached to a method in the editor.

3.5 A design for an introductory refactoring tool

In designing a front-end to a refactoring tool for introductory students, the challenge is to find a balance between supporting the functionality we want and the simplicity that the students require. We want the full set of refactorings to be available, but we want it to be obvious which refactorings apply in different locations.

A simple change to the IDEA popup menu would be to dynamically disable and enable menu items depending on the context of where the cursor is (see Figure 10). However, this would leave the student attempting different combinations of cursor placement and text selection in order to try to "unlock" the menu item. This solution may be worse than leaving the design as it is.

Our design involves augmenting an editor with an improved understanding of the elements that make up a source file. By making the source code of methods an "entity" in the editor, we can then attach refactoring operations to it using a drop down menu.

Figure 12 – The popup menu attached to a class in the editor.

3.5.1 Methods as user interface objects

A key to the BlueJ interface is that key abstractions in the system are represented on-screen as objects with operations. The current BlueJ interface applies this technique to the main display of classes and objects, but does not attempt to represent any abstractions other than source in the editor. To make methods in the editor a distinct user-interface element they should be visually distinct. However, we do not want the user to lose the overall feel that they are editing a single source file. By slightly changing the background colour of the area in the editor representing each method, this effect is achieved (see Figure 11).

We now want to make all the method level refactorings available from the method interface entity. A drop down menu box attached to the method area allows access to the refactorings. This drop down menu could either be placed on the left hand side of the editor (perhaps in the column that some source editors reserve for breakpoint information, line numbering, etc.), or could float in the text at the top right corner of the method (see Figure 11).

The drop down menu would only show those refactorings applicable to a method. Because the "extract method" refactoring is only applicable when a portion of the method is selected, it would be greyed out in the drop down menu unless a suitable area is selected in the editor.

3.5.2 Classes as user interface objects

In BlueJ, a UML diagram of the classes in the system is the basic tool for interacting with a project. However, as we saw with methods, classes do not have any special

user interface presence in the editor view. Unlike a method, which is generally only a small portion in the editor, the source for a class is generally the entire file that is being edited. Therefore, it does not really make much sense to highlight the class source by changing its background colour, as this would affect the whole source file. Instead, just the head of the class should be distinctly coloured. This is shown in Figure 12.

Similar to methods, classes would have a drop down menu box displayed at their header. It would show only those refactorings applicable to a class.

3.5.3 System wide undo

The BlueJ editor has an undo facility that removes the most recent changes to a source file. However, when performing refactorings, changes are made simultaneously across many source files in the project. Therefore, adding refactoring support involves redesigning undo to work at a higher level than it does at present.

Our approach would be to have a special undo window. It would track all refactorings, changes made in the editor, and test cases executed (see chapter 5 for our design of integrated testing support). It many ways it would be like the Adobe Photoshop history/undo window [McClelland2002]. The Adobe Photoshop history window stores a growing stack of operations that have been performed on a loaded picture, with the ability to go back to any previous state by selecting it from the list. In Figure 13 we see where the "paintbrush" and "eraser" operations have been undone by the user selecting the "crystallize" operation in the history stack. The "paintbrush" and "eraser" operations can be redone by clicking on them in the stack.

For BlueJ the operations that are tracked would be large-scale changes to source. So, for instance, if a user added some code to a method, an operation is not added for each keystroke in the editor, rather an operation is added such as "code changed in method X". Refactoring operations are added with details of the type of refactoring and where it was performed. Even operations that do not result in the

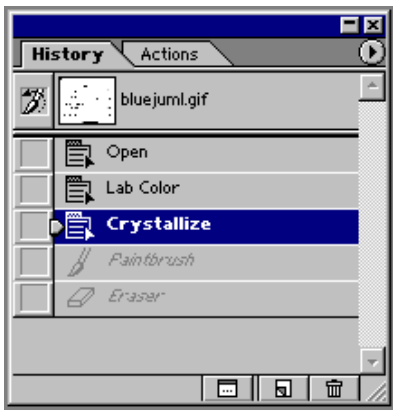

Figure 13 – The History/Undo window in Adobe Photoshop.

changing of source code could be added, such as an entry like "Unit tests run failed:5 succeeded:2".

At any point the student could go back to any previous point in the undo stack by clicking on the operation's entry in the undo window. The undo stack is not saved with the project – it only exists during a single session of using BlueJ. Obviously, there would be the facility to commit all the changes made by saving them to disk.

A comprehensive undo facility is valuable for a student refactoring tool because it will encourage experimentation with refactorings. If students know that they can perform refactorings, run unit tests and compile and interact with the system, yet be assured they can always go back to a previous working state it will greatly support their experimentation.

3.5.4 Summary

A design for this refactoring extension to BlueJ has been completed. Implementation of this design has started by successfully integrating the transmogrify back-end with BlueJ, which can now be used to perform simple refactorings. A full prototype implementation of this design will be completed shortly.

4 Chapter 4

TESTING

This chapter discusses testing and its role in introductory teaching. First we examine testing in general and why it is important to software development. Secondly we look at the position of testing in introductory courses and approaches that have been developed for introducing it to the early computer science curriculum. Next we look at the practical techniques of testing that are needed by students and examine each one for its relevance and suitability in a teaching context. Finally we look again at the tools that are available for performing testing and conclude by asking whether a tool could be developed which better facilitates the teaching of some of these testing activities.

4.1 Why test?

Because software plays an ever larger part in our lives, an increasing emphasis is being placed on software reliability. Testing is an important facet in ensuring software reliability. In fact, testing is recognised by industry as an important part of software development and a significant proportion of the resources devoted to software development are consumed by testing activities.

Analysing the reasons why testing is being conducted allows us to structure testing activities into categories, each of which achieves a different purpose. This categorisation is independent of the scope or method of test selection which, as we shall see later, both give us alternative ways of categorising testing. [Pan2002] [Hetzel1988] suggests the following testing categories:

- Correctness testing
  Determining if the software behaves "correctly". This is the category that is predominantly being referred to when people refer to software testing;

- Performance testing

  Determining if the software performs within resource limits. Even though software may not have explicit performance limits there are implicit limits such as it cannot take an infinite amount of time;

- Reliability testing

  Takes a variety of measurements to estimate the probability that the software is correct. These measurements can be derived in part from the results of correctness testing on the system. This form of testing also looks at the robustness of a system. Robustness is its ability to handle exceptional inputs or stressful environmental conditions and continue working (though not necessarily produce a correct result for these inputs); and

- Security testing

  Testing software for the purpose of stopping external parties accessing resources on a system through flaws in the software. Security testing may be performed by simulating security attacks.

Of course, there are other possible categorisations and sub-categorisations, though these four categories are generally regarded as the major types of testing. These categories are useful for understanding why we test, but do not help us understand the mechanisms of how we can test. To do that we must look at categorising testing according to scope and test selection [Whittaker2000]. When we look at the scope of testing we are categorising the testing based upon the constituents of the system that are being tested. The scopes are:

- Unit testing

  Testing individual software components without regard to the rest of the system. This may require the construction of code that exists solely to emulate the behaviour of the system in a known way so that the units can be placed in the environment they need without depending on other parts of the system;

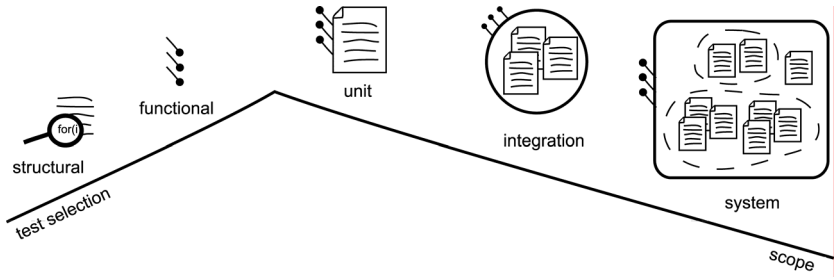

Figure 14 – Two orthogonal classifications of testing.

- Integration testing

  Testing multiple components together that have each received prior unit testing. The focus of this integration testing is mainly on the boundary of the components or those sections that represent communications between components; and

- System testing

  Testing a collection of components as a completed product. This concentrates on whether the system satisfies the overall application functionality that was the goal of the project.

Test selection determines the types of tests that are to be used. If tests are chosen without considering the internal structure of the component being tested then it is called functional testing. If tests are selected based on knowledge of the implementation of the component then this is called structural testing. It is important to note that purpose, scope and test selection are orthogonal (see Figure 14); we may do structural unit testing in order to check performance criteria or we might equally do functional system testing to check for correctness.

4.2 Testing in education

It is disappointing that despite its importance in software development, it is difficult to fully explore testing as a topic in introductory computer science courses. The load on students in mastering so many other fundamental topics of computer science often squeezes a thorough treatment of testing out of the limited time available to introductory courses. The first, and sometimes only, exposure to testing

may be the basic use of a symbolic debugger in early lab classes and practical work. Often the formal introduction to testing is left for software engineering classes later on in the curriculum [Shaw1991].

The importance of testing in computer curricula has been noted for many years. Software Validation is listed as a core unit in the *Computing Curriculum 2001* [ACM2001], and the *Guidelines for Software Engineering Education* [Bagert1999] lists testing as objectives of both the Software Quality[1] and Software Construction/Evolution modules. Even a much earlier computing curriculum contains testing as an important objective [ACM1991].

Whilst teaching testing is not common in practice, there have been several proposals for introducing testing into introductory curricula. One group proposes the introduction of software engineering concepts early in the curriculum and thereby introduces testing as one of these software engineering techniques. The second group has proposed various schemes for introducing just the testing discipline to introductory courses. We will firstly look at those who favour the early software engineering approach.

4.2.1 The early software engineering approach

Hilburn [Hilburn2000] has argued for a number of years for more of a focus on software quality[2] at the undergraduate level. He believes that software quality "addresses a central and critical issue in the development of computer software" and that faculty "… do not devote enough attention to teaching their students how to develop high-quality software" [Hilburn2000 page 167]. His proposal uses a software engineering model known as the V Quality Model (a variation on the traditional waterfall life cycle model) to be a conceptual framework for developing a curriculum based on quality. Software processes such as the Personal Software Process (PSP) and the Team Software Process (TSP) are introduced across the

[1] Listed as software verification and validation

[2] Quality here refers to both the usefulness of the product for the customer and also to how well the process used to develop the product has worked. That is, it is a measure of both the product and the process.

whole undergraduate curriculum [Hilburn1997]. These processes help the students to analyse and improve not just their designs, but also to look critically at their own software processes and attempt to improve upon them. PSP emphasises quality review at every stage of the software life cycle and provides for collection and analysis of metrics to measure quality [Fekete2000].

Although Hilburn acknowledges the importance of testing, he strives to emphasize the quality process over testing techniques so as not to develop students who test only in an ad-hoc, trial and error fashion. Testing in the V Quality Model involves two separate testing passes. In the first pass, testing frameworks are developed for system testing, then integration testing and then unit testing. These frameworks are developed in conjunction with requirements analysis, architectural design and detailed design respectively. The second pass is in the reverse order and involves execution of the testing strategies developed. A quality review is required to move from one test stage to another.

Another approach to introducing software engineering early in the course is that taken by Jackson and McCauley who have used the establishment of documentation and design standards as a framework for teaching software engineering principles and techniques across the undergraduate curriculum [Jackson1997]. The document framework has the following components: requirements documentation, design documentation, implementation documentation and verification/validation documentation. Students are required to submit solutions incrementally and each component is marked promptly so that feedback is received by the student before the next documentation component is started. Once the students are used to the document framework, subsets of it are used to introduce various software engineering concepts such as characteristics of good software, maintenance and software testing plans and techniques. For instance, students may be given the design documentation and code of another group and then be assigned the task of structurally testing the code. The results of a comparative study of the performance of later year students on project work showed that students who had been exposed

to the software engineering concepts through the introduction of this documentation framework obtained grades higher on average than those who had undertaken a more traditional introductory curriculum [McCauley1998].

Both of these proposals require major changes to the structure of an introductory course and also introduce costs such as a greater load on teaching assistants who must mark and return documentation before students proceed onto later stages. In the following section we discuss other less radical proposals which plan to introduce testing techniques in the early curriculum whilst still leaving formal software engineering training to later years.

4.2.2 Early testing

Several different approaches to the inclusion of testing into introductory programming courses have been presented over the last few years. Some of these introduce testing methodology by requiring students to submit test plans or test logs which are marked by teaching assistants. Other approaches aim to give hands-on experience in the practice of testing.

Jones [Jones2001] suggests some testing activities that can be incorporated into introductory courses. These include:

- students grading other student's programs using their own test data; and
- instructors providing programs with known bugs and assigning marks for discovering bugs and documenting the bug discovery process.

Goldwasser [Goldwasser2002] proposes a simple scheme to augment existing programming assignments with the principles of software testing. Each student submits both source code and a test set for the assignment and these test sets are run against all other submitted assignments from the rest of the class. A portion of the student's grade is based on how well the student's test set uncovers bugs in the other students' assignments. Despite being quadratic in the time taken to run all the

tests against all the solutions (this is not a large problem if the test execution is automated), the scheme has the following advantages:

- the competitive angle of each student trying to write tests to find flaws in their friends' code provides a level of fun;
- students who may be struggling to complete the implementation of a solution may still feel part of the exercise because they can still write test cases; and
- students' tests will be run on a diverse set of implementations.

Kay [Kay1994] suggests providing the students with automated testers as part of a comprehensive electronic submission system. The system incorporates some initial feedback to the student at submission time regarding the program's performance on a set of public tests, and reporting for teaching assistants of the program's performance on a set of private tests.

None of the schemes mentioned above explicitly deals with testing in object-oriented programs. Whilst many of the techniques for testing procedural programs are applicable in object-oriented coding, the shift to object-oriented in introductory teaching has added new testing problems [Barbey1994].

One of these problems is that the overhead for the construction of test cases is high in some object-oriented languages, with a new class needing to be declared for holding test cases. Some object-oriented languages such as Python provide solutions to this problem by including the ability to easily include a test harness and launcher to test on a class by class basis.

Another problem in some object-oriented languages is that scope and access levels checks can prevent test classes from calling the application methods which need to be tested (for instance, in Java if a test class is defined in a different package to the class being tested, then methods with "package", "private" or "protected" level

access cannot be called by the test class). Of course, there are easy solutions to these problems for experienced programmers, but at first year level, solving them may require the introduction of language constructs that are not appropriate at that stage of the course.

Another problem caused by object orientation relates to the size and number of separate units that require testing. Procedural programming tended to produce large monolithic applications with long function definitions. Whilst it was hard to construct good tests for these, the infrastructure required to set up and run the tests was relatively straightforward. Object oriented programming tends to produce better separated units of code with smaller and more precise methods [Ferret2002]. This means that testing can be more effective because methods tend to do only one thing, but the sheer number of tests means that some sort of testing infrastructure is needed and hence tools to help manage the testing process are now more important.

4.3 Testing techniques for students

At a professional level, testing is a formal discipline with a large body of theory and terminology behind it. For introductory teaching, we need to view it at a much simpler level and see it as a set of techniques that help to achieve more reliable software. Different techniques need to be applied at different stages in the development process and for these techniques, tool support can be useful for both ease of use and to help with understanding concepts.

Students gain practical skills by utilising testing tools. It is a general principle of university level education that whilst it is valuable to learn specific skills, it is also important to learn the general concepts behind a skill so as to be able to transfer the knowledge to new and different domains. This principle applies to the use of tools in introductory software engineering. So for example, learning how to use the symbolic debugger in Visual Studio is a useful skill, but it is important that students grasp the concept of using a symbolic debugger in general so that they can transfer

the knowledge to the use of a debugger in a completely different environment such as `gdb` under Unix.

Unfortunately, some of the tools for testing that are currently used in education are inappropriate. This is because professional tools have a level of sophistication and complexity that is not appropriate for a learner. Other techniques have to be taught without any tool support because tools in that area are at an early stage of development, or non-existent.

The stated goal of this thesis is to design better tools for software engineering in introductory education. To this end we need to know what the different types of activities are that our tools should support. We have identified the following types of activities in relation to testing as being representative of those performed currently:

(1) Testing immediately after implementation

After producing an implementation for a unit of code, tests are written to exercise the unit and to verify that it behaves correctly. This may be done through test drivers, small snippets of code that execute the application code and generate results. Often these test drivers are thrown out after use or are intertwined with the application code and are uncommented as needed. A more permanent form of this type of testing is to write test drivers in a separate class with the aim of keeping the test code and rerunning it later. There are no tools required to perform this form of testing as the test drivers can be written in the same language as the application code. It was typical that this type of test was not fully automated, i.e. the test execution generated results that were inspected manually for correctness. Increasingly common though is to use a testing framework such as JUnit [JUnit2002] to organise the test drivers and to collate and verify their results automatically.

Another way to do this testing is the use of a tool such as Blue [Kölling1999] or BlueJ [Kölling2001a] that allows the interactive construction of objects and execution of method calls from within a development environment. This facilitates the testing of application code immediately without setting up test drivers, but the tests performed are not recorded and must be redone in their entirety to check the code again. In particular, setting up the objects to be tested may be a time consuming operation and as this needs to be repeated for each test, it acts as an impediment to the students actually performing the testing

In situations where test drivers are written, testing after implementation may be seen as identical to regression testing (discussed below in case (3)). However because some environments such as Smalltalk, Scheme and BlueJ also offer the more interactive forms of testing discussed above, we have separated (1) and (3) to include those environments in this discussion.

(2) Testing after detecting a bug

Bugs can be detected through the construction of test drivers as in (1) above, or perhaps through user feedback and general system testing. However, detecting that there is a bug does not necessarily help locate the bug or indicate how to fix it. After a bug is detected a different form of testing is performed to elucidate the location of the bug and analyse behaviour of the program in the area around the bug. The tests may be as simple as adding some print statements to the code or may involve the use of a symbolic debugger. The use of print statements is a good technique because it does not require any tool support and is applicable no matter what language is being used. Using a symbolic debugger allows some advanced functionality such as breakpoints and single stepping to better understand the behaviour of the code. Symbolic debuggers also allow the inspection of the state of an object which may help understanding of its behaviour. Blue provides an alternative technique for inspecting object state without necessarily using a symbolic debugger [Rosenberg1997].

(3) Testing after fixing a bug

Once a bug is detected and fixed it is important to make sure that the bug is not reintroduced later on in the development cycle. A comprehensive set of test drivers similar to those described in (1) are useful to ensure this. Test cases are developed that exercise the once buggy code and make sure it is behaving correctly. This style of testing, usually called regression testing, may be automated so that it is possible to execute the tests regularly throughout the development cycle. A tool such as TestMentor [Silvermark2002] helps in this automation by providing facilities for comparing expected and actual results and collating a report about the execution of the tests. Testing frameworks such as JUnit are often also used for this form of testing.

At an advanced level there are tools to help analyse the effectiveness of test cases. These code coverage tools [Connell1996] inspect the execution of test cases and report on the percentage of application code paths that have been traversed by the tests. This can then lead to the development of more effective test cases.

(4) Testing before implementation

Another technique advocates the construction of test cases before the corresponding application code is written. This style of development is called test driven development (TDD) [Beck2002]. Test cases are constructed but unlike in (1) and (3), the tests are designed to fail on the current application code. That is, the tests are not written to prove the correctness of an implementation, or the correctness of a bug fix, but are written to show the absence of the correct implementation. Code is then constructed to make the tests work and once this is done, the cycle restarts with the construction of more test cases. In this development style the design of the test cases leads the design of the code. It is argued that this leads to systems with better cohesion and looser coupling [Beck2002]. Whilst TDD is achieving excellent results and may become an important

development methodology there are caveats to concentrating solely on it for first year teaching. These include:

- despite its popularity, many of the success stories for TDD are still anecdotal and future research may show limitations with the test-driven approach; and

- because of the relative infancy of TDD, techniques for teaching it have not even begun to be developed. In fact, it is quite possible that some of the tenets of TDD are unsuited to introductory level programmers and that more traditional programming techniques need to be taught first. A recent paper exploring the teaching of some of the new, so called, agile methodologies such as TDD concluded that it "cannot be properly appreciated until you've suffered the pain of alternative heavy weight methods or indeed no methods at all" [Lappo2002 page 38]. It may be that without experiencing the pain of non-agile development methodologies the advantages of TDD are not obvious.

  Of course, we do not want to discount TDD either so it is important to also consider the tool support for performing it. Currently TDD is performed using test frameworks such as JUnit but without any other form of tool support.

Another group falling into this type of testing is the extreme programming (XP) community. Whilst not relying on construction of tests before any functionality, XP programming tries a similar philosophy with tests writing intertwined with the writing of the code with the goal being to build a working minimal system first. In this way it is like a milder form of TDD. There is increasing interest in the possibility of using XP in introductory courses [Allen2003].

Identifying these different types of testing activities allows us to design educational testing tools that support testing activities in early programming courses. The

availability of such tools can increase the likelihood of students actively performing testing activities.

4.4 Current tool support for testing

The four testing activities (1)..(4) identified in the previous section are often performed with, or aided by, the use of testing tools. Some of these tools are designed explicitly for use in educational environments, whilst others have been developed for use in professional programming environments. We will briefly look at the advantages and disadvantages of some sample tools to see if the professional tools are capable of being used in an educational setting and to see if the educational tools available cover the range of testing activities that we have identified.

Whilst looking at these testing tools it is helpful to examine the four stages that execution of a test normally involves. The initial stage is the setting up of the test - either identifying and locating the source code to be tested or constructing objects that the test will be performed on. The second stage is the actual writing and construction of the test itself. The third stage involves executing the test and the final stage involves the validation of the results. Each tool approaches these stages differently. Some stages are automated, making them easier to perform repeatedly and quickly. Other stages are performed manually, providing more flexibility but at the cost of increasing the time required to execute the stage.

4.4.1 Symbolic debuggers

Symbolic debuggers have existed since the early days of modern computing [Kernighan1984] but for most of that time the debuggers in use did not evolve much in terms of functionality or features. A reason suggested for this is that debuggers are so specific to a particular machine, operating system and language that improvements to debuggers on one platform were often not transferred to new machine architectures unless programmers were willing to re-implement the improvements on the new architecture [Ramsey1992]. Recent years however have seen a rationalization in the number of architectures and operating systems and now

symbolic debuggers all have an improved set of core features such as breakpoints, source level code display and data inspection [Zeller1996].

Tests in a symbolic debugger are set up manually by indicating breakpoints in the source code. The construction of the actual test itself is normally performed in an ad hoc manner through the use of single stepping and breakpoints to examine the execution of sections of the code. The results of the test are viewed manually by inspecting the debugger display and comparing to expected values. Although using them is a manual and labour intensive task, symbolic debuggers are, at least in principle, simple to use and therefore are suitable for use in first year education in support of testing activity (2).

4.4.2 Unit testing with JUnit

Unit testing has undergone a revival in the last few years with the adoption of extreme programming (XP) development techniques such as refactoring and pair programming [Jeffries2000] [Beck1999]. Part of the revival is due to XP's emphasis on testing as a means of ensuring correctness of refactorings, but the revival is also due to the introduction of a testing framework for Java called JUnit [JUnit2002]. Whilst XP provided a compelling *motivation* for unit testing, JUnit provided a standard method of *performing* unit testing and hence lowered the start up costs of introducing testing to a project (previously developers would often construct their own testing framework). The JUnit framework has since been adapted to many other languages and now there are unit testing frameworks for languages such as C++, Python and C#. Because of the similarity between all the unit testing frameworks, we will concentrate the rest of this discussion on the Java version, JUnit.

Talbott defines four important components of XP style unit testing [Talbott2001]:

- unit testing is a *practice* - it is not effective unless the developers make a conscious effort to do it;

```
public class EmailTest extends junit.framework.TestCase
{
    Email testMail1;

    public EmailTest(String name) { super(name); }

    protected void setUp()
    {
        testMail1 = new Email("From: Andrew <andrew@some.com>\n" +
                              "To: Joe <joe@another.org>\n" +
                              "Subject: that memo\n\n" +
                              "Just wanted to remind you\n");
    }

    public void testHeaderParse()
    {
        assertTrue(testMail1.hasSubject());
        assertEquals(testMail1.getSubjectField(), "that memo");
    }

    public void testHeaderAddition()
    {
        testMail1.addHeader("X-List", "mailman");
        assertEquals(testMail1.getField("X-List"), "mailman");
    }
}
```

Figure 15 – A sample of test code written using the JUnit framework.

- unit testing is about *verification* – the tests make checks that they expect to succeed;
- unit tests focus on *behaviour* - not how something is implemented but how it should behave; and
- unit tests focus on the *external interface* - this usually involves testing the public methods of a class.

The set up phase for the JUnit testing framework consists of writing code that makes available to tests a set of sample objects, known as fixtures. Fixtures are arbitrary Java objects that the programmer writes code to construct. JUnit ensures that these objects are recreated before each test to ensure that there is no side effect to the order in which tests are performed.

The construction phase of the tests involves the programmer writing test methods in standard Java and using some predefined methods for asserting test results. Tests

are then executed automatically through the framework and the results are collated and displayed in a user interface.

In the example (shown in Figure 15), we are constructing tests for a piece of code that deals with email objects. We start by extending our test class from `junit.framework.TestCase` thereby gaining access to a whole set of assertion statements that will determine the success and failure of each test method. Along with `assertEquals()` and `assertTrue()`, which we have used in the sample code, there are numerous other JUnit supported assertions such as `assertSame()` and `assertNotNull()`. The `setUp()` method, executed before each test method, is run to ensure that our fixture objects are in a known state before being acted on. There is also a corresponding method called `tearDown()` that is executed on completion of each test. We then invoke JUnit, passing in our test class (or a `TestSuite` object which defines a collection of test classes).

The JUnit framework comes with a variety of extensible interfaces for displaying results. One standard output class is the `TextRunner` which displays the results to standard output. An alternative is the `SwingRunner` which displays the results using a GUI. Whichever display class is chosen, the JUnit framework will display for each test case (every method with a name starting with `test` is considered to be a test case) the status of the assertions, and, if any failed, provide a stack trace showing the expected values for the assertion that failed. The results of executing our sample code with the `SwingRunner` interface are shown in Figure 16.

In summary, the JUnit framework is an important step in the renaissance of the *practice* of unit testing throughout the general programming community. Whilst the framework itself is not large and could be reimplemented quite easily, its acceptance in the programming community has meant that it has become a de facto standard for unit testing. JUnit provides an excellent structure for the construction of test cases and execution of those tests.

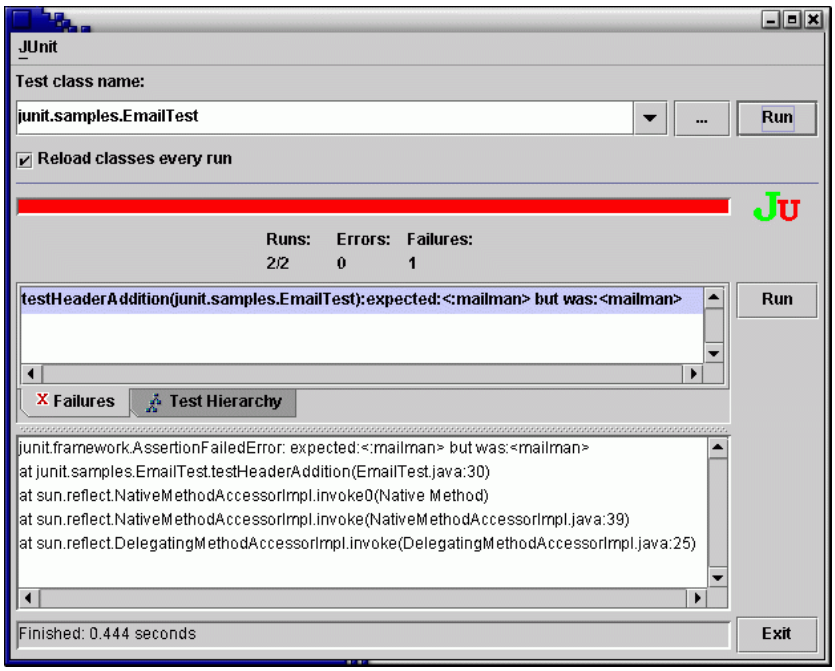

Figure 16 – The SwingRunner showing the result of the EmailTest.

4.4.3 TestMentor

Test Mentor by SilverMark software [Silvermark2002] is a tool that automates testing of components. Early versions were Smalltalk based but they have now introduced a Java Edition. Test Mentor takes design models, actual object interactions and the static class structures and relationships, and uses them to generate nearly complete tests. It is suitable for the construction and execution of regression tests as in testing activity (3). The generated tests are either represented in an internal structure or can be represented as standard Java source. The advantage of the internal representation is that it can be interacted with through a GUI so that non-developers can create and run tests without knowing how to program in Java. The following sections outline the approach taken by Test Mentor.

4.4.3.1 Construction of test assets [1]

In testing object-oriented programs, the construction of objects on which test stimuli are to be applied is often time consuming. Test Mentor provides a "simple means to define and reference reusable assets that embody strategies for

[1] Test Mentor uses the term assets for what unit testers would call a test fixture.

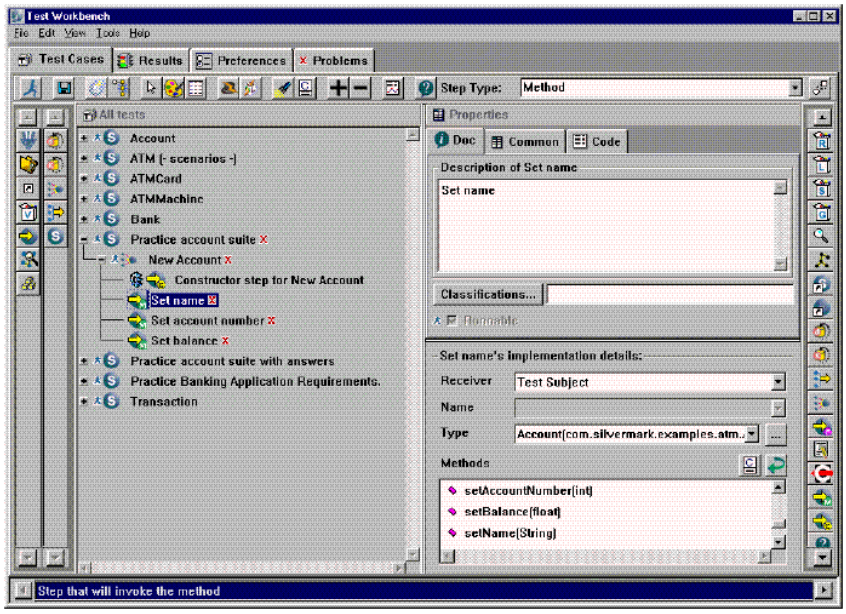

Figure 17 – Construction of "steps" in Test Mentor (reproduced with permission from [Silvermark2002 page 92]).

instantiating and configuring objects" [Silvermark2002 page 88]. Test Mentor provides explicit support to help in the initial phase of test construction by allowing a set of shareable objects to be developed. Test Mentor terms these shared objects assets.

The approach Test Mentor uses to construct assets is to build a set of "steps". These steps represent the instantiation of an object or the application of methods to an object, but are labelled with free form strings. These free form string descriptions can then be used by testers to construct assets without necessarily being able to program in Java.

To illustrate the construction of assets with Test Mentor we follow an example presented in the user manual. Firstly, a new asset is created by selecting the class and constructor to use in the GUI. Then a description of the steps needed to construct the asset are entered. These are of the form of statements such as "set name", "set account number" etc. For each of these statements, Test Mentor constructs a test step that then links to the actual Java methods needed to perform the step. So for instance the "set account number" step can be linked to the

`clearNumber()` and then `setAccountNumber(int)` method calls. Any parameters that are required by a method can either be specified immediately or generalized such that they are asked for when the asset is constructed. The entire procedure for constructing assets is performed with a GUI and the methods that can be selected at each step are dynamically extracted from the classes in the system (see Figure 17). One advantage of this approach is that "steps" can be constructed by programmers who are familiar with the language being tested, but assets and tests can be constructed from these "steps" by specialised testers who do not know the target language.

4.4.3.2 Construction of test stubs

Test Mentor aids the second phase of performing testing by partially automating the initial construction of test cases. This automation merely provides a starting point for the construction of test suites, but it is claimed that this automated test generation gives the user "a good head start on your test creation and a starting point for further test creation" [Silvermark2002 page 137]. Test Mentor uses some novel techniques to determine an initial set of test stubs. These include:

- taking a Rational Rose [Boggs1999] sequence diagram and constructing tests that emulate the sequence of method calls indicated in the diagram;
- working through the public methods of an object and generating a set of tests that exercise each of these methods. These tests can be generated in different styles including a style of test where after each method call the state of the object is compared to a known reference value. This aggregate validation is discussed in the next section; and
- generating tests based on actual interactions of the objects in the system when the program is run. A user can select a class and then tell Test Mentor to monitor all of its interactions for a particular class. The tests that will be generated are based on actual calls made to the object during the course of the execution (see Figure 18).

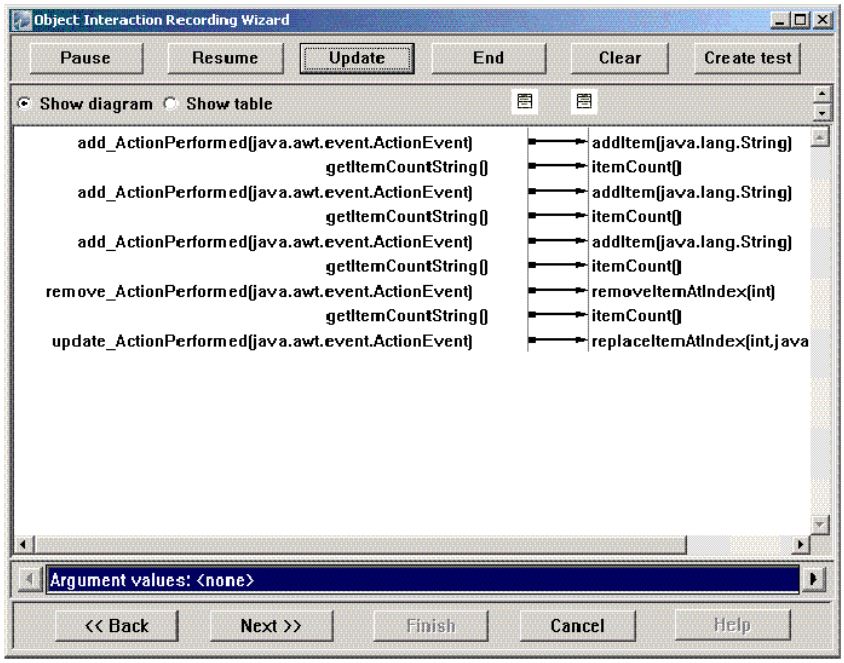

Figure 18 – Recording object interaction with Test Mentor (reproduced with permission from [Silvermark2002 page 84]).

For each of these, it must be noted that whilst a set of test stubs is generated, the tests are not yet complete and the user must still fill in literal values for method parameters and add checks to make sure that the objects are in the desired states.

4.4.3.3 Validation

After the construction of assets and the generation of test stubs, the tests still need to have validation added to them. The validation phase of each test subjects the assets to stimuli and then validates that the components end up in an expected state. Test Mentor provides facilities for comparing the actual state of an object with the expected state. Some of these facilities are:

- a family of `assert()` calls that compare expected values to actual values and throw exceptions if they are not the same;
- a validation "step" that can be specified using the GUI. This step takes a reference to an actual value, a reference to an expected value and then compares them using a comparison operator such as less than or greater

than. The comparison policy can be specified programmatically allowing the implementation of custom validation steps; and

- an aggregate state "step" that extracts the property values from an object and collates them into a list of values. This aggregate state list can then be compared to the expected list of values of a known reference object. These known reference objects are referred to as gold standard objects.

4.4.3.4 Summary

Test Mentor is an example of the tools that are available for professional testing activity. There are many other similar tools with different languages as the focus and with slightly differing functionality. What they have in common is a user interface for the construction of test cases and for the automatic execution of these test cases and the collection of the results. The target for some of these environments is the non-programming tester – someone who works in the quality assurance department of a company but who does not need to have particular experience with the language that a project has been developed in. For such people, a user interface for the generation of assets and tests that hides the underlying language details can be invaluable.

However, while it can be used by a non-programmer, the functionality it provides is still quite advanced. That is, whilst the non-programmer may not be familiar with the programming language being used, they are still professional testers and hence are familiar with more concepts and practices from the testing field. Furthermore, they also have the time to invest to become familiar with a complex tool such as Test Mentor.

The ability to generate test stubs quickly using reflection and call monitoring can be very useful for large legacy applications, but this is not an activity that introductory students are likely to be performing.

4.4.4 BlueJ

The BlueJ environment is an integrated development environment designed explicitly for introductory teaching [Kölling2001a]. BlueJ provides a unique object interaction facility that can be used for testing activity (1). In this short section we will examine this object interaction and its current application in testing in more detail. A more comprehensive look at the complete tool will be presented in section 5.1.

The object interaction mechanism in BlueJ allows the initial test set up phase to be performed by the user instantiating objects and placing them on a workbench called the object bench. Method calls can be made on these objects, with the results either being displayed as text or resulting in an object. If an object is returned it can be placed on the object bench and then interacted with.

The beauty of BlueJ is that tests can be performed on classes immediately after the code has been constructed. No test harnesses or test drivers are required to execute the methods that have just been constructed. However, this testing is ephemeral. In particular, the potentially time consuming set up phase, where objects are constructed, must be repeated after each compilation. This acts as an impediment to using the tool to test methods.

Also, because the interaction is not recorded, the construction and execution phase is totally manual. Tests cannot be easily repeated to confirm the behaviour later on in the program development, and there is no automated checking of the results of a test to confirm the behaviour is as expected. This prevents BlueJ from being used to support regression testing as in testing activity (3).

In summary, the object interaction features of BlueJ make it easy to perform quick testing of methods with a great deal of flexibility. It is not ideal, however, because the tests performed are totally manual and cannot be automated.

4.5 Summary

In this chapter we have looked at testing and its importance in introductory education. Despite a great deal of work in incorporating testing into introductory education we believe that there is still a lack of adequate tool support suitable for first year students. When looking at tool support, we have identified four activities as being representative of the type of testing performed by students. Each of them is listed below:

(1) Testing immediately after implementation

Partly supported by testing frameworks though the overhead of constructing test drivers is an impediment. Also supported in an efficient manner by interactive environments such as Smalltalk or BlueJ, although these tests are transient;

(2) Testing after detecting a bug

Adequately addressed by symbolic debuggers and tools that allow object inspection;

(3) Testing after fixing a bug

Supported by testing frameworks such as JUnit. Professional tools such as Test Mentor were examined but are too complex for use by students; and

(4) Testing before implementation

Uses testing frameworks such as JUnit, but has no other tool support.

The next chapter proposes a design for a tool that will add support for testing activities (1) and (3), and to a lesser extent (4). The tool incorporates the quick and efficient BlueJ object interaction with the regression testing facilitated by JUnit, to provide an easy way for students to construct and run test cases. We will show that when added to the already existing BlueJ support for object inspection these facilities can provide a tool that covers the gamut of testing activities that are required for introductory education.

5 Chapter 5

DESIGN OF TESTING SUPPORT IN AN EDUCATIONAL INTEGRATED DEVELOPMENT ENVIRONMENT

This chapter introduces the BlueJ development environment as a platform to be used for the inclusion of testing tool support. The new testing facilities that have been added to BlueJ are shown via a walk through of a typical testing task. Other features of the testing support are also discussed. We begin with some background on BlueJ and its predecessors.

5.1 Blue

The Blue project involved[1] the development of an integrated object-oriented language and object-oriented development environment designed explicitly for introductory teaching [Kölling1999]. Blue addressed the need for a development environment suitable for teaching object-oriented programming to first year students. The key feature of the language Blue was that all language concepts were clear and simple, yet it supported modern functionality such as pure object orientation, garbage collection and pre and post conditions. The Blue development environment had an interface that was simple and easy to use, but still included most of the important software tools required for a student; a compiler, a project manager, an editor, a class browser and an integrated debugger. It also had novel functionality that allowed the students to visually instantiate and interact with objects in the system.

The Blue environment and virtual machine was written in C++ and was only supported on Solaris and Linux. A Windows port of the Blue environment was commenced but never completed.

[1] Whilst the Blue system is still available for download, it is no longer under active development and we will therefore refer to it in the past tense.

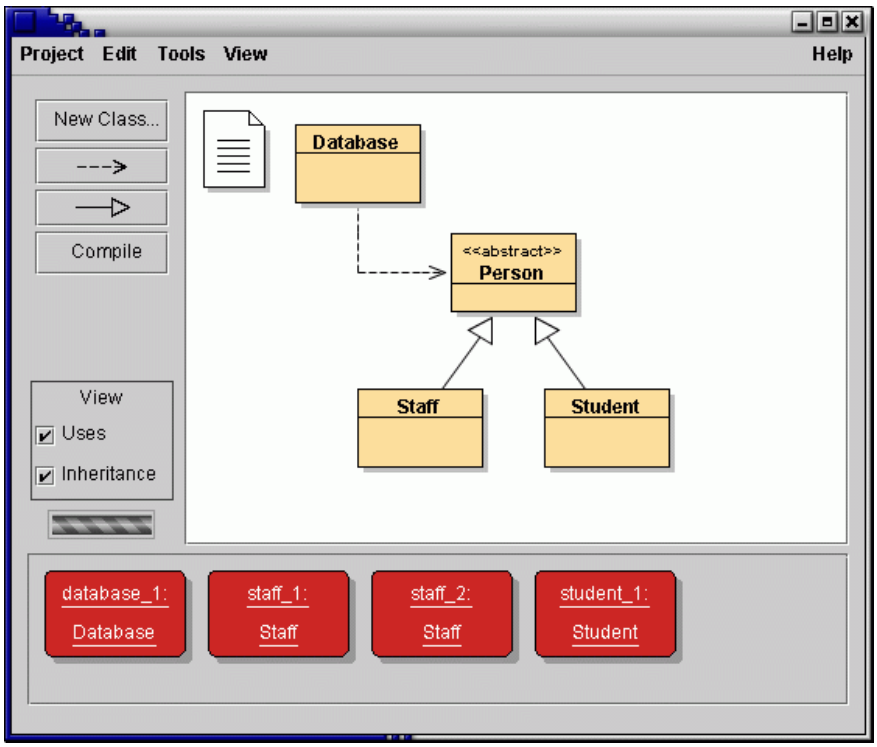

Figure 19 – The main BlueJ window showing the UML style class diagram and objects on the object bench.

5.2 BlueJ

The popularity of the Java language and its suitability for introductory teaching called for an environment similar to Blue to be built with support for Java. This work commenced in 1998 and resulted in a system known as BlueJ (originally JavaBlue).

The BlueJ project provides support for Java in an environment with most of the features of the Blue environment [Kölling2001a]. Because BlueJ is itself written in Java, it can be run on many more platforms than Blue could.

The main features of note in BlueJ are its support for UML style class diagrams, its direct interaction with objects, its object inspector and integrated debugger, and its support for Javadoc. These are discussed in more detail below.

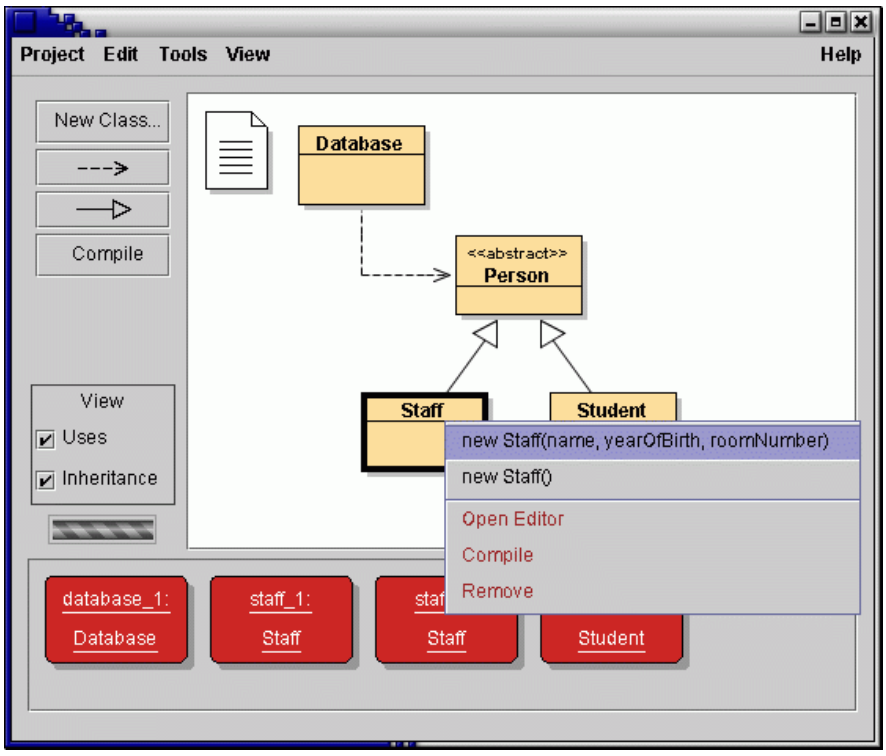

Figure 20 – The popup menu of a class in BlueJ.

5.2.1 UML style class diagrams

The main display of BlueJ is a simplified UML diagram of the classes in the system. Each class is displayed as an icon with different styles of shading to indicate compilation status and with UML stereotypes to indicate different class types such as "«abstract»", "«interface»" or "«applet»". There are two different relationships between classes that are shown on the diagram. Inheritance relationships are shown with a solid lined arrow and references between classes are shown as dashed lines. The relationships can be specified by interacting with the diagram (which creates the corresponding relationship in the source) or can be specified by editing the source (which automatically reflects back in the diagram). Figure 19 shows the main display of BlueJ.

Each of the classes displayed has a popup menu (selected by right clicking on the class – shown in Figure 20) that can be used to compile the class, open the source editor (double-clicking on the class is a short cut to this operation), remove the class

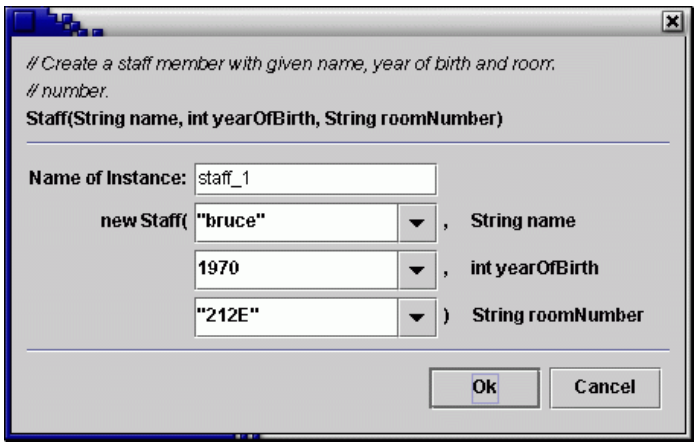

Figure 21 – Parameter passing when constructing an object in BlueJ.

or interact with the constructors of the class. The popup menu displays all public constructors for the class, as well as all public static methods defined in the class. Figure 21 shows the dialogue that is displayed when one of these constructors is selected. The user is first given a chance to input any parameters to the constructor, and then a representation of the constructed object will be created on the object bench at the bottom of the main display. The user interaction with these objects will be discussed in next section.

Objects on the object bench are transient. They are automatically removed if any change is made to their corresponding class or if the project is closed.

5.2.2 Direct object interaction

Direct object interaction is a unique feature of the BlueJ environment. The objects on the object bench have popup menus that display the public instance methods for objects of that class. Methods from each of the objects' supertypes are also displayed in cascading popup menus. As with constructors, if a method takes a parameter and the method is selected from the popup menu, the user is presented with a dialog to enter parameters to the method. If the method returns a result, this result is displayed to the user. For primitive types, BlueJ displays the result as a string, but for all other types BlueJ allows the returned object to be placed on the object bench where it can also be interacted with.

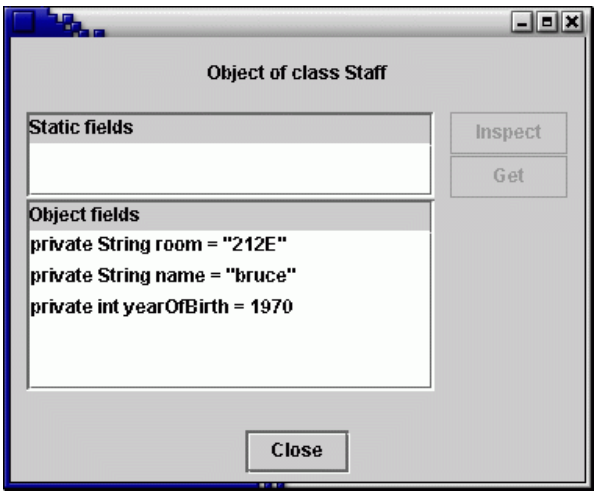

Figure 22 – Inspecting an object.

The direct interaction facility is valuable to students because they can experiment with objects without writing any code. It allows teachers to structure an objects-first introductory course where students can play with classes and objects before seeing any source code [Kölling2001b]. This direct interaction, where objects and classes are treated as first class entities in the system, also helps reinforce the concepts of object-oriented programming i.e. the one to many relationship between classes and objects, instance methods operate on objects, not on classes, etc.

5.2.3 Object inspection

BlueJ allows the user to view the internal state of objects on the object bench (see Figure 22). The inspection dialog shows both the instance and static fields of an object. Fields of a primitive type are displayed as strings. If a field is an object reference, then the object that it refers to can also be inspected. This facility of BlueJ allows students to explore the structure of objects without having to use a symbolic debugger.

5.2.4 Integrated debugger

BlueJ has a symbolic debugger that supports breakpoints, stack inspection and single stepping through source code (see Figure 23). Whilst it is relatively simple to use, the debugger contains concepts that may not be appropriate for beginner

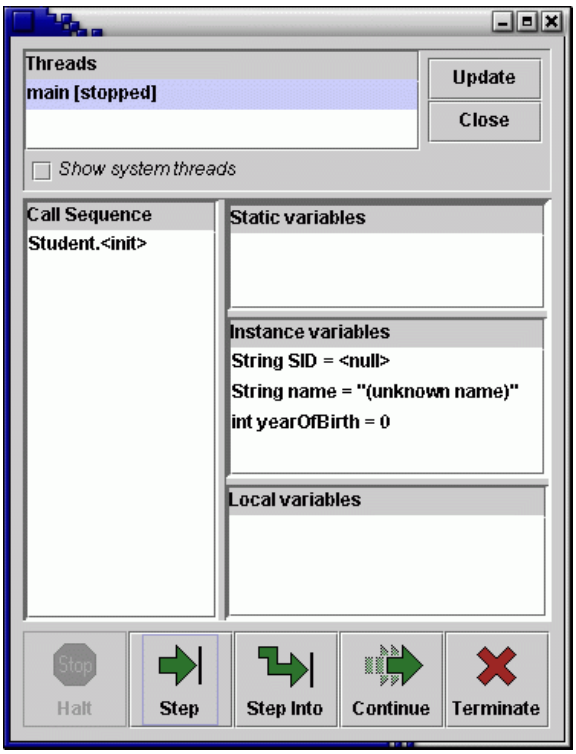

Figure 23 – The BlueJ debugger.

programmers such as threads and call stacks. The BlueJ debugger also suffers from some problems with its robustness in unusual situations. An effort is underway to improve the debugger in BlueJ, including looking at different models for debugging threaded programs [Schulz2000].

5.2.5 Javadoc generation

Whilst BlueJ can show all the classes in a single package as a UML diagram, it is desirable to be able to browse all the classes in a project. The standard Java way of providing information about all classes in a project is to generate Javadoc documents. Javadoc specifies a way of placing method and class documentation into the source code of a class and then generating a set of hyper linked web pages detailing this information for browsing in a web browser. BlueJ supports the launching of the Javadoc tool and the launching of a web browser to view the resulting web pages for the current project.

5.3 Introduction to testing in BlueJ

The testing functionality we have built into BlueJ incorporates the object interaction ability already existing in BlueJ with the ability of JUnit to perform regression testing. We have seen the need for this in the previous chapter where we analysed various testing tasks. We surmised that testing tasks (1) and (3) are particularly deficient in tool support. Unit testing frameworks such as JUnit support a standard way to write tests but do not provide any tools that help in this task. The BlueJ interaction mechanism is useful for task (1) but does not provide any recording facility, so tests must be redone by hand, eliminating its usefulness for task (3).

When adding unit testing to BlueJ it was important that the unit testing functionality did not impact upon students who were not using unit testing. To this end, unit testing is integrated into the BlueJ interface in an unobtrusive way. If a project is not using testing then the testing functionality barely impacts upon the user experience. The addition of the "Test" button on the main display and a "test runner" dialog, whose display is toggled from the View menu, are the only plainly visible testing functions. Consideration is also being given to a preference setting that disables all testing functionality in BlueJ (including removing the "Test" button and any other menus relating to testing) for situations where it is not appropriate for students.

5.4 Testing overview

The most fundamental functionality provided by the unit testing extension to BlueJ is the recognition of JUnit tests as a special type of class in the BlueJ system (see Figure 24). The following functionality has been incorporated into the BlueJ user interface in order to support unit test classes:

- ability to construct a test case class;
- ability to run all the tests in a test case;
- ability to run an individual test method from a test case;
- ability to move the test fixture from a test case onto the object bench;

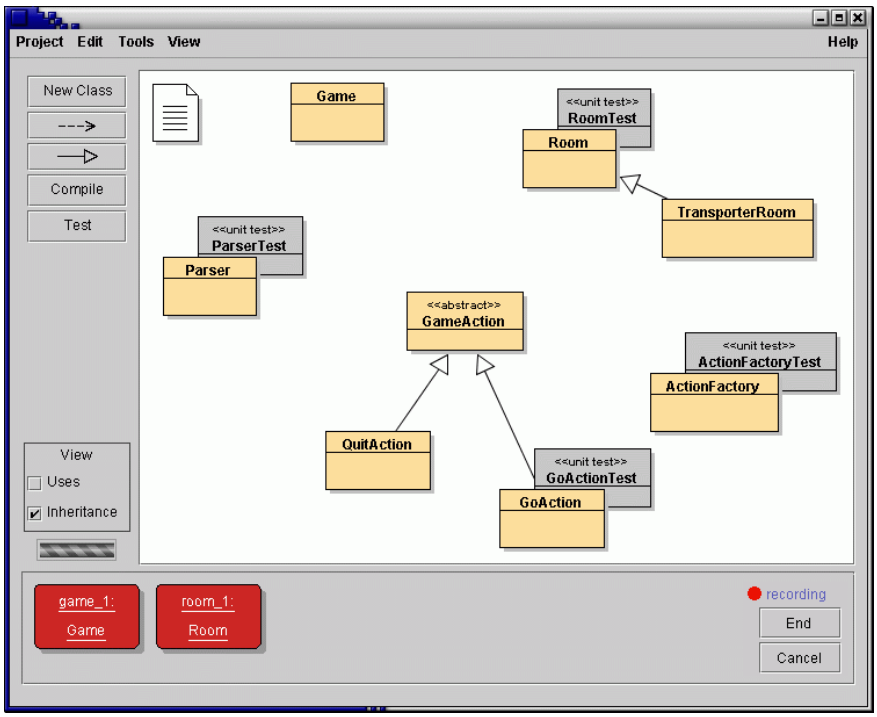

Figure 24 – The BlueJ system showing the addition of the unit testing functionality.

- ability to construct a test fixture from objects on the object bench; and
- ability to construct a new test method in a test case by interacting with objects on the object bench.

If we look at Figure 24, we can see the three user-interface changes that are present in the unit testing version of BlueJ. Firstly, the grey classes on the UML diagram represent the unit tests. An additional button labelled 'Test' has been added to the buttons along the left edge of the main window. This button runs all the tests present in a project. More details of the running of tests can be found in section 5.5.5. In the bottom right hand corner on the object bench, buttons have been added to end and cancel the recording of tests. An explanation of this test recording functionality is in section 5.5.3.

A more in-depth explanation of all the testing functionality is presented in the following sections by introducing a walkthrough of testing using a conventional

testing methodology. The chapter will end with a discussion of the use of the BlueJ testing support in handling testing activity (4) – test driven development.

5.5 Conventional testing walk through

In order to describe the new testing functionality that has been integrated into BlueJ, we will walk through the process of implementing tests on a *zork* [Spear1994] style text adventure program. This assignment is a modified version of that presented in [Kölling2001b], with a looser coupling between the parser class and the rest of the system.

The *zork* game contains four major classes (or groups of classes). The main `Game` class in the original assignment has been refactored into `GameAction` objects that encapsulate the behaviour of a particular command such as "go" or "quit". `GameAction` objects are returned from an `ActionFactory` that maps command strings to the action's class. The `Game` class itself is now mainly a loop that mediates between the other classes. The game consists of a loop that instructs the `Parser` to read commands from standard input and then moves the player between locations. Locations are represented by `Room` objects, which contain a description and a set of exits that indicate in which of the four standard compass points one may leave. The set of classes in the assignment can be seen in Figure 24.

In the following walkthrough, we will use the terminology *target class* to refer to a class in BlueJ that is the being tested by a unit test class. For instance, if we had a class called `FooTest` that primarily tested `Foo` objects, the class `Foo` would be the target class.

We will assume that we have already constructed the `Parser` class. It contains two methods `tokenizeAndLower(String)` and `tokenizeAndLower(Reader)`. The first tokenises a string, lowercases the tokens and then returns the tokens in an array. The second method does the same as the first except using the `readLine()` method from the `Reader` as the input source.

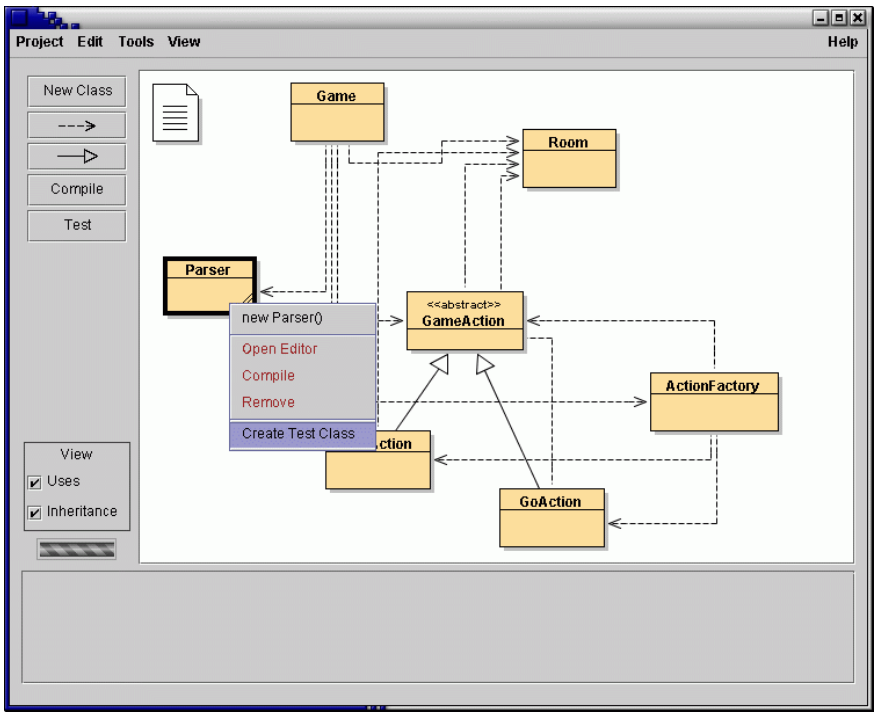

Figure 25 – The popup menu for creating a new test class.

5.5.1 Recording of Ad-Hoc Test Interaction

At its most fundamental level, the functionality that is to be shown in the following walkthrough is that it allows the recording of normal BlueJ object interactions as unit tests. This allows the combination of ad-hoc testing that BlueJ has always supported to be combined with regression testing. Section 5.5.3 shows the details of this interactive construction of a unit test method. Section 5.5.9 shows how the BlueJ ad-hoc interaction can be used to create a unit test fixture.

5.5.2 Constructing the test class

We wish to start some testing so as our first step we construct a unit test for the `Parser` by selecting "Create Test Class" from the `Parser` class' popup menu (see Figure 25). Unit tests are represented in the interface in the same way as other classes in the system albeit with a UML stereotype of "«unit test»" and a distinctive grey colour. The resulting unit test class will be automatically named `ParserTest`.

The decision to represent unit tests as first-class user interface objects was a fundamental design decision. The alternative would be to build all the testing functionality into the standard class object (perhaps on their popup menus) and to hide the existence of the JUnit testing classes. We chose to present test classes to the user because we believe that the introduction of JUnit testing is about more than just providing additional functionality to the student. BlueJ should also reinforce concepts of testing. This is similar to the way BlueJ reinforces the concept of objects and classes by presenting them as two different types of user interface elements in BlueJ. By presenting unit test classes as separate entities, the distinction between the class being tested and the test class is reinforced. If the JUnit classes were hidden in the system, students would feel that the test cases are some sort of magic, where as in reality they are relatively straightforward pieces of Java code.

The generated unit test class is constructed using a template unit test file that comes packaged with BlueJ. The template can be customised by the user or system administrator to fit in with any local requirements of coding style. An alternative approach that was considered was to make BlueJ construct the test class as a set of stub test cases that are based on the methods existing in the target class. This obviates the laborious task of creating many test method stubs when we are testing a large target class that already exists. There are numerous systems in existence that have this functionality such as the JUnit extensions for NetBeans [Netbeans2002]. This approach was not selected for BlueJ because it was felt that the use of BlueJ in introductory teaching means that it is rare to have to deal with large amounts of legacy code, and furthermore, that automatically creating empty test stubs may give students a false impression that they are constructing actual tests rather than just test stubs.

There is nothing in a standard unit test class that ties it to a particular class or set of classes in the system. However, classes are by far the most predominantly used "unit" for testing in Java and hence the unit test will often be testing a single class. Therefore, when a unit test is created from a target class, BlueJ assumes that there is

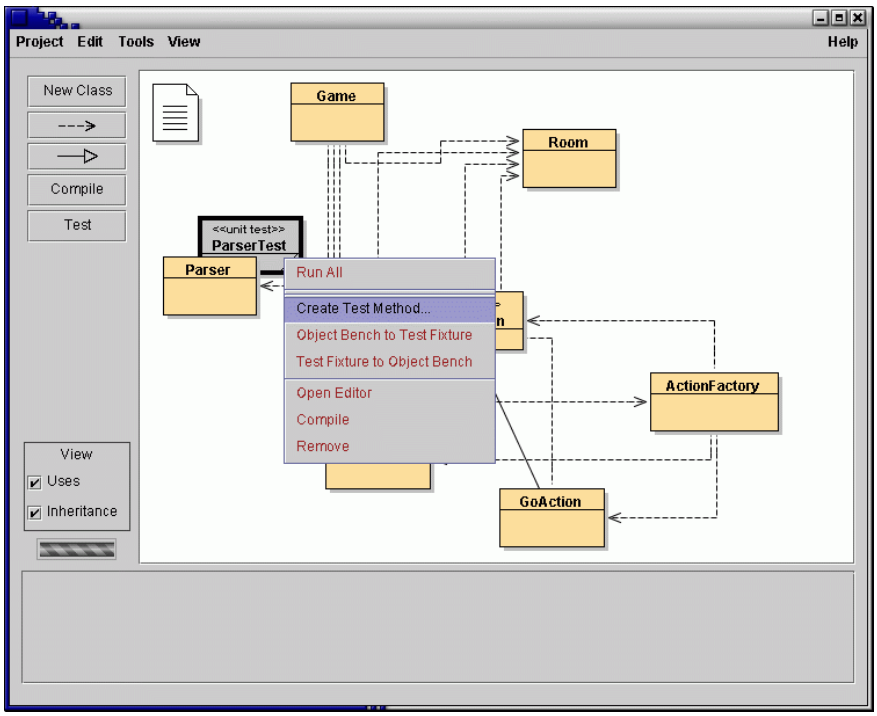

Figure 26 – The popup menu for creating a test method.

a tight coupling between the two classes and will keep them together on the diagram. This "association" between class and unit test class is used to determine special compilation dependency rules as will be seen in a section 5.5.12 of this walkthrough.

5.5.3 Creating a test method

The first test we wish to create is a test to see that the `Parser` class is always returning a valid `String` array, no matter what the input. We start the construction of a new test case by selecting "Create Test Method…" from the unit test class' popup menu (see Figure 26). We are prompted for the name of the test and we enter the name "NotNull". An "end test" button appears at the bottom right corner of the object bench. All our interactions with BlueJ, from now until the "end test" button is pressed, will be recorded in the `testNotNull()` method of the `ParserTest` class.

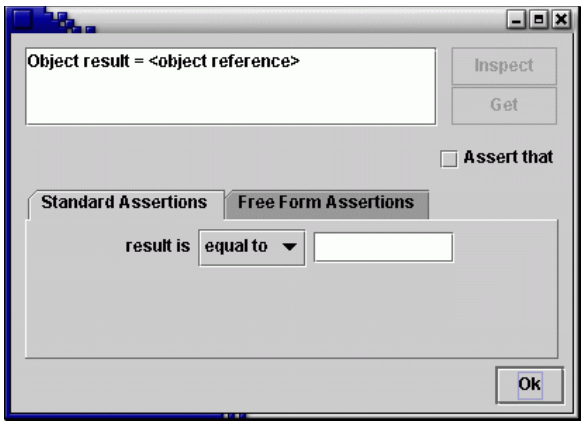

Figure 27 – The result and assertion dialog.

We construct a new `Parser` object by selecting "new Parser()" from the Parser class' popup menu. This creates a `Parser` object and places it on the object bench. We can now execute methods on the `Parser` object and BlueJ will perform the operation. We select the `tokenizeAndLower(String)` method and are presented with a dialog asking for parameters to the method call. As we are testing to make sure the method always give us a non-null string array, we start with a boundary case like the empty string "". As with normal BlueJ interactions, a result dialog is now displayed showing the returned object. However, because we are in test mode, the result dialog is extended to show an assertion panel that can be used to make assertions in the current test. The assertion panel is shown in Figure 27.

5.5.4 Asserting results

We want to assert that the result we received from the method is not null. To do this we check the "Assert that" checkbox. We can then select the type of assertion that we want from the drop down list of supported JUnit assertions. In our case, we select the "not null" assertion. When we click on the "Ok" button, this assertion is added to the current test. We repeat this process for some other cases we wish to test such as the strings "a" and "AA ab". As a final test we execute the `tokenizeAndLower(String)` method, passing in null as a parameter. The code is actually being run on a second virtual machine so we will get an exception thrown when attempting to use the null pointer. BlueJ catches the exception and highlights

```
public void testNotNull()
{
  Parser parser_1 = new Parser();
  {
    String[] result = parser_1.tokenizeAndLower("");
    assertNotNull(result);
  }
  {
    String[] result = parser_1.tokenizeAndLower("a");
    assertNotNull(result);
  }
  {
    String[] result = parser_1.tokenizeAndLower("AA ab");
    assertNotNull(result);
  }
}
```

Figure 28 – The unit test source of a method created through BlueJ interactions in the ParserTest class.

the line in the Parser class that the exception occurred at. Because we get no result dialog when an exception is thrown, we do not get any chance to make any assertions about the result. For the time being we will have to leave off testing for this exception. We revisit the testing of exceptions in section 5.5.15.

After exhausting all the cases that we wish to check we click the "end test" button in the bottom right corner of the object bench. The `testNotNull` method has now been added to the `ParserTest` class (its source is shown in Figure 28). After compiling the test, we are now ready to run it.

5.5.5 Run All

The "Run All" popup menu item runs all the tests in the selected test case. The results are displayed in a dialog indicating the list of test methods and whether they passed or failed. A large bar is displayed in green if all the test methods pass. If there are any failures then the bar is displayed in red. In a case where the test has failed, the bottom half of the dialog displays the results of the assertion, showing the line where the assertion failed and what the actual and expected results were. An example of this test dialog is shown in Figure 29.

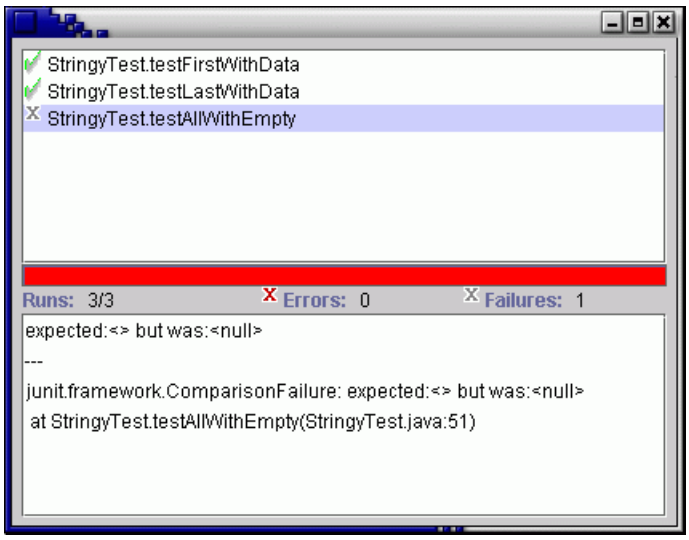

Figure 29 – The dialog showing the result of running three tests.

BlueJ utilises its own implementation of the standard JUnit `SwingRunner` user interface. The BlueJ implementation adds the ability to jump from an assertion failure to the line of code in the test case that caused the failure by double clicking. The BlueJ version differs internally from the normal `SwingRunner` in that the interface code and the execution of the tests occur on different virtual machines. This will be discussed in more depth in the implementation section 6.6.

5.5.6 Dealing with arrays

We have successfully constructed a test method for the Parser class that ensures the `tokenizeAndLower(String)` method always returns a non-null string array. We would now like to construct a test ensuring the actual values returned in the array are accurate. We start in the same manner we did for the previous test by selecting "Create Test Method…" from the test class' popup menu. This time we will call the test "Basic". We construct a `Parser` object and execute the `tokenizeAndLower(String)` method with a parameter of " AA Ab bb". From the resulting dialog, we click on "Inspect" to view the returned object. The string array is then displayed as in Figure 30.

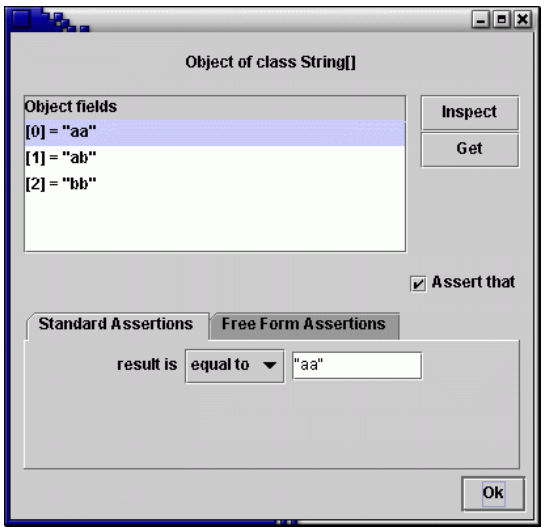

Figure 30 – The result and assertion dialog for an array.

The difference between arrays and normal method results is that BlueJ cannot add a single assertion when the "Ok" button is pressed because there are multiple elements in the array and therefore there could be multiple assertions required. To handle this, when an array result is being viewed, each array element has its own "Assert that" checkbox. When an array element is selected and the "Assert that" checkbox is checked, the assertion type and value are particular to that array element.

As a time saving device, when the "Assert that" checkbox is checked, the value of the actual result from the corresponding array element is copied into the assertion value text field. So for instance, when we click on array element result[1] and check "Assert that", the string "ab" is copied into the assert value text field. If the current implementation had generated the incorrect result it is easy to indicate the correct value by changing the text field.

For each array result we check the "Assert that" checkbox and then select "Ok". We then click the "end test" button and BlueJ constructs a `testBasic()` method that contains all the assertions. The source of the resulting `ParserTest` class is

```
public void testBasic()
{
  Parser parser_1 = new Parser();
  {
    String[] result = parser_1.tokenizeAndLower(" AA Ab  bb");
    assertEquals(result[0], "aa");
    assertEquals(result[1], "ab");
    assertEquals(result[2], "bb");
  }
}
```

Figure 31 – The unit test source of a basic method created through BlueJ interactions in the ParserTest class.

shown in Figure 31. The details of how the tests are constructed from the object interactions is discussed in the implementation section 6.4.

5.5.7 Testing using standard Java classes

The final test we wish to make for the `Parser` class is a test for the `tokenizeAndLower(Reader)` method. Our first step is to construct a `Parser` object on the object bench. Secondly, we need to construct an object that satisfies the `java.io.Reader` interface so we can pass it as a parameter to the method. Luckily there is a standard Java class called `java.io.StringReader` which can be constructed from a string and which satisfies the Reader interface. We can construct a `StringReader` object using the "Use Library Class…" facility of BlueJ. Firstly we start the recording of a new test case named "BasicRead" using the "Create Test Method…" menu option. We then select "Use Library Class…" from the "Tools" menu. In the resulting dialog we enter `java.io.StringReader` and press return. A list of all the constructors for the `java.io.StringReader` class will appear. As there is only one, the `java.io.StringReader(String)` constructor will be highlighted. When we now press "Ok", the normal object construction dialog of BlueJ will appear. We pass in the string "Go East" as the parameter to the `StringReader`. It will now appear as an object on the object bench. This can be seen in Figure 32.

The `StringReader` on the object bench can be interacted with using the same BlueJ mechanisms as the `Parser` and other objects we have dealt with. For

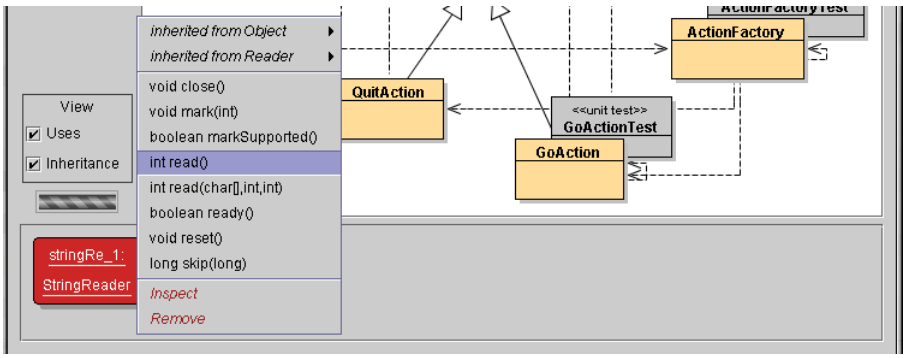

Figure 32 – A `java.io.StringReader` object on the object bench. The popup menu shows the method calls which can be made on the object.

instance, we could call the `read()` method or inspect the fields of the object. However, in this case we have constructed the object to use as a parameter to the `tokenizeAndLower(Reader)` method.

We select the method from the Parser object's popup menu and the method call dialog is displayed. We select the text field for the parameter to the method and then we click on the `StringReader` object. BlueJ will check that the object is of a compatible type, and if it is, will enter the name of the `StringReader` object in the method call. When we select "Ok" this call will then be executed and the resulting array will be displayed. We make assertions about this result in the same manner that we did earlier in the walk through. The resulting test's source code is shown in Figure 33.

All Java classes can be constructed and used in this way. For instance, if a method dealt with Java collections then `LinkedList` objects or `HashMap` objects could be created on the BlueJ object bench and used in testing.

5.5.8 Sharing test objects

In constructing tests for the `Parser` class we notice that there are some objects that we use in each test (a `Parser` object for example). It would be useful to be able to share the effort of constructing the objects between all the test methods. JUnit also has its own concept of objects which are shared between tests. The set of these objects in a test case is called the *test fixture* and are instance variables which are

```
public void testBasicRead()
{
  StringReader reader_1 = new StringReader("Go East");
  Parser parser_1 = new Parser(reader_1);
  {
    String[] result = parser_1.tokenizeAndLower(reader_1);
    assertEquals(result[0], "go");
    assertEquals(result[1], "east");
  }
}
```

Figure 33 – The unit test source for a method created using the Java StringReader class.

created in a designated method in the test case called `setUp()`. A natural fit between BlueJ and JUnit would be if the object interaction methods of BlueJ could be used to construct the test fixtures in standard JUnit tests (in the same way that the BlueJ interaction is being used to construct JUnit test methods). Similarly, if JUnit test fixtures could be brought into BlueJ, then object interaction and test generation could be performed with existing JUnit tests. The unit testing features of BlueJ have this ability.

5.5.9 Creation of a test fixture

To illustrate the construction of a test fixture we will construct some tests for the Room class. The purpose of this class is to represent the locations between which the player can move in the *zork* game. There are four exits that a room can have (north, south, east, west) though it is not necessary that each room has all of the named exits. Exits are specified using the `setExits(Room north, Room south, Room west, Room east)` method, with a null parameter for a direction meaning that there is no `Room` accessible through that exit.

Our first step it to construct a test class for `Room`. We select "Create Test Class" from the `Room` class' popup menu and the new `RoomTest` class is created. We would like to make a small set of connected rooms to test that rooms can be successfully navigated. We construct three room objects on the object bench with descriptions of "west room", "east room" and "south of east room" respectively. On the object bench these objects have the names room_1, room_2 and room_3.

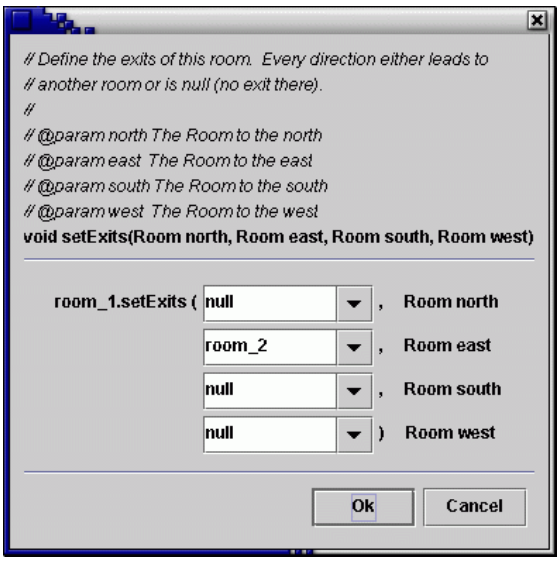
// Define the exits of this room. Every direction either leads to
// another room or is null (no exit there).
//
// @param north The Room to the north
// @param east The Room to the east
// @param south The Room to the south
// @param west The Room to the west
void setExits(Room north, Room east, Room south, Room west)
room_1.setExits (null , Room north
room_2 , Room east
null , Room south
null) Room west
Ok
Cancel

Figure 34 – The method call dialog executing Room's setExits() method.

We use the method interaction facilities to call the `setExits()` method on each room instance, passing in the other room objects as parameters. BlueJ extracts method header comments from the Java source and displays this in the method call dialog. It also displays the method's variable names, which is useful in this case to distinguish between the four room objects that the `setExits()` method takes. The method call dialog is shown in Figure 34.

We now want to store the three Room objects on the object bench as shareable objects in our `RoomTest` class. We select "Object Bench to Test Fixture" from the `RoomTest` class' popup menu. The source code to construct the objects is saved into the `RoomTest` classes' `setUp()` method. The objects which have been saved now disappear from the object bench. They can be restored in two different ways as explained in the next section.

5.5.10 Restoring a test fixture

A test fixture can be restored to the object bench by selecting "Test Fixture to Object Bench" from the test class' popup menu. This will execute the `setUp()` method and place the constructed objects onto the object bench. Users can interact

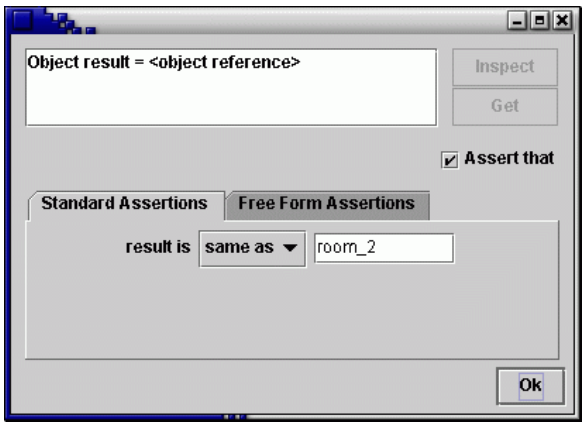

Figure 35 – The result and assertion dialog when the object returned is already on the object bench.

with these objects on the object bench identically to the way that they handle objects that have been constructed through normal interaction.

The other method of restoring a test fixture to the object bench is by creating a test method. When a test class has a test fixture, the fixture's objects are automatically placed on the object bench whenever the recording of a test method is started. Any method calls which are made on these objects are local to the test method being recorded and will not impact upon the state of the test fixture objects in other test methods.

We can see the restoration of a test fixture in action by constructing a test method for the `Room` class. We select "Create Test Method…" from the test class' popup menu. We give the test method the name "RoomExit". The three room objects that we created as the test fixtures for this class now appear on the object bench. We select the `nextRoom(String)` method on the room_1 instance and enter "east" as a parameter. The resulting dialog is shown in Figure 35. We note that whenever an object reference is given as a result, BlueJ attempts to see if this object is already on the object bench. If so, the name of the object on the object bench is automatically placed in the assertion text field and the type of the assertion is set to "Same As". We can see this in Figure 35 where the result of the going "east" is the room_2

object on the object bench, which is therefore automatically copied in as the assertion value.

5.5.11 Extending a test fixture

We may not always know exactly what objects we would like in a test fixture when the test class is first constructed, so it is necessary that BlueJ has the ability to add to objects in a test fixture. We can extend a test fixture by selecting "Test Fixture to Object Bench" and then constructing new objects or calling methods. When the objects on the object bench reach the new state we wish for them, we then select "Object Bench to Test Fixture" from the test class' popup menu. BlueJ will now rewrite the `setUp()` method of the test class with the current object bench. To prevent accidental deletion of a test fixture, a warning message is displayed when attempting to replace a test fixture with objects from an object bench that was not initially constructed from the same set of objects.

5.5.12 Silent compilation

When a class is compiled, BlueJ will automatically attempt to compile the corresponding unit test class. This compilation is attempted silently – any errors will not be presented to the user with the usual "hilight line in the editor" technique. When a test class fails to compile, an error message is displayed in the status bar and the class is left with an uncompiled state appearance in the class diagram. This should provide enough feedback to the user that something is wrong in the test class. However, if a user selects "Compile" from the unit test class' popup menu then it is assumed that the user wants this class in particular to be compiled and hence is interested in any error messages. In this case, BlueJ presents the test class' compilation errors to the user in the standard way.

5.5.13 Tests created outside of BlueJ

A design philosophy of BlueJ is that it must be able to work with standard Java projects taken from other sources. BlueJ augments the projects with a file that stores details such as the arrangement of the class diagram or details such as which test is associated with which class. However, if this information is lost or is not

available, BlueJ must still work without it. For this reason, the unit testing support has the facility to deal with unit test classes that are not associated with any other class.

A test class can become disassociated from its target in a variety of ways. Test classes can start out disassociated if they are created through the "New class…" dialog (as opposed to selecting "Create Test Class" from the target classes' popup menu). BlueJ also allows the importation of existing Java classes using the "Add Class…" dialog – an existing unit test class could be imported in this manner.

A disassociated test class can be associated with another class by selecting "Associate" from its popup menu. A dialog is then displayed showing the classes in the system that allows the user to select one for association. Only test classes that are not already associated have this menu option displayed.

When a test class is left disassociated, all that it loses is the diagram coupling and the benefits of automatic compilation when the target class is compiled. Other than this, they function identically to associated unit test classes.

5.5.14 Run individual tests

Whereas traditional IDE's only allow interaction with an application in a monolithic way (by executing the main method of the program), BlueJ allows interaction at an object, class and method level. The standard JUnit interface only allows the execution of all tests in a test class (although JUnit does allow test suites to be created, containing individual tests from multiple test classes, this must be done programmatically).

The popup menu for a unit test class in BlueJ has a menu item for each test defined in the class. By selecting a test method from the popup menu, just a single test method is run. If a test is successful then a simple message indicating the success is displayed in the status bar at the bottom of the screen. If the test fails then the result is displayed in a dialog showing the failed assertion, similar to the dialog

```
public void testException()
{
  Parser parser_1 = new Parser();
  {
    try
    {
      parser_1.tokenizeAndLower(null);

      fail("NullPointerException should have been thrown");
    }
    catch (NullPointerException success)
    {
    }
  }
}
```

Figure 36 – The unit test source for a test method generated when an exception is caught.

shown by the "Run All" menu. This allows quick execution of specific tests, which is useful when editing the particular area of code that those tests target.

5.5.15 Testing exceptions

In section 5.5.4 we passed null to the `tokenizeAndLower()` method which generated an exception. Normally, when an exception is generated and is not caught by any user method, BlueJ catches the exception and displays the line of source code that threw the exception. Because no result dialog is displayed, the user misses the opportunity to make an assertion about the result. To deal with this, when BlueJ is in test method construction mode and an exception is encountered, BlueJ will generate test method code that ensures that an exception is thrown. The basic pattern of the code is to wrap the method call in a try/catch block, indicating success when an exception is caught, and to fail if the code reaches the end of the try block. An example of this is shown in Figure 36.

We should make the point that any choice would have been valid for the decision on what to do when `tokenizeAndLower()` is passed a null parameter value. It could have been programmed to return null in the case where it was unable to process tokens, rather than allowing it to throw an exception. We then would have constructed a test to assert that when a null value is passed into it, we do get a null as the return value. An important aspect of the construction of unit tests is that they

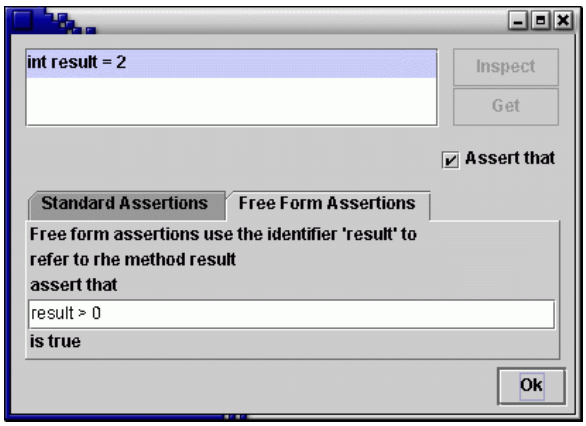

Figure 37 – The free form assertion dialog.

act as a form of documentation for the code. An explicit test of its behaviour when passed a null dispels any ambiguity about how clients should expect the code to behave in that situation.

5.5.16 Free form assertions

There are some rare cases where the default JUnit assertions cannot express an assertion the way a programmer would like. An example of this would be if a programmer wanted to assert that a method result was within some range. To support this, BlueJ allows the programmer to insert free form assertions. This is a tabbed panel on the assertion dialog that allows entry of a free form text string. The free form string (which can refer to "result" in order to reference the return value) should be a boolean expression, and is inserted directly into an `assertTrue()` assertion statement in the resulting test method (see Figure 37).

5.5.17 Further ideas

There are some other potential features of the unit testing extension that have been considered but which have not been implemented because it is not clear what their interface would look like, or there are still technical problems remaining with their implementation. One of these would be to allow assertions that are more powerful than just single expressions – one would like to be able to assert that all results from a particular function satisfy a given function *f(x)*. Another useful feature would be

to have an interface that allows the construction of tests that cover a wide scope of input values. For instance, the user could specify a range of input values rather than needing to call the tested method by hand for each value.

5.6 Test driven development

Test driven development (TDD) was first discussed in section 4.3. It is a development style that encourages the construction of test cases *before* the corresponding implementation is coded. TDD is a new style of development and it was not initially considered when the design for the unit testing in BlueJ was first developed. However, as we will see, the unit testing support in BlueJ is compatible with the TDD methodology.

5.6.1 Walkthrough

Let us imagine that we are still working on the *zork* example. We wish to use TDD to develop a new type of `Room`, one that has more than the standard north, south, east and west exits of a regular room. We will call our new room `TransporterRoom`. We create the new `TransporterRoom` class by selecting "New Class…" from the edit menu. Using the dependency arrow button, we add an inheritance dependency between the class `Room` and `TransporterRoom`. We now select "Create Test Class" from the `TransporterRoom` class' popup menu to create a `TransporterRoomTest` class.

We double-click on the `TransporterRoomTest` class and the standard BlueJ editor window opens showing us the source code to the unit test. All the unit test classes we have developed in the first part of this walk through can similarly be edited just like standard BlueJ classes.

The idea of TDD is that we start by constructing a test that fails. First we need to decide what functionality and interface we would like our `TransporterRoom` class to have. We write the test under the assumption that our `TransporterRoom` will be able to be implemented using the interface we select. We do not allow the implementation details to affect how we would like to `TransporterRoom` to

```
public void testPortalRoom()
{
  TransporterRoom troom = new TransporterRoom("transporter room");
  Room destroom = new Room("somewhere to go");
  troom.addExit("portal", destroom);

  assertSame(destroom, troom.nextRoom("portal"));
}
```

Figure 38 – The unit test source for a TDD method in TransporterRoomTest.

behave. If it turns out we have created an unimplementable interface, we can back up a few steps and try a different interface with a new understanding of how the implementation needs to be done in practice.

We decide that the functionality we would like is to be able to add an extra exit to the room. The parameters for this should be the name of the exit and the room that the exit leads to. We construct our test "PortalRoom" by constructing a new method called `testPortalRoom()`. We write code to construct a `TransporterRoom` object and a `Room` object that we can use in this test. We then make a call to `addExit()` on the `TransporterRoom` object. Finally, we make an assertion that the next room through our exit we have just constructed is indeed the correct room. The source code to this test is shown in Figure 38.

We now click on the "Compile" button in the editor. The compilation fails because there is no String constructor for the `TransporterRoom` class. We add a constructor and try to compile the test again. It fails again because there is no `addExit()` method. We add an empty `addExit()` method, taking a `String` and a `Room` as parameters and try to compile the test again. The compilation succeeds! We can now run our test by selecting the "Test" button in the editor window for the `TransportRoomTest` class. The "Test" button is identical to selecting "Run All" from the test class' popup menu, but does not require the programmer to switch back to the main BlueJ window and therefore is a convenient shortcut (see Figure 39).

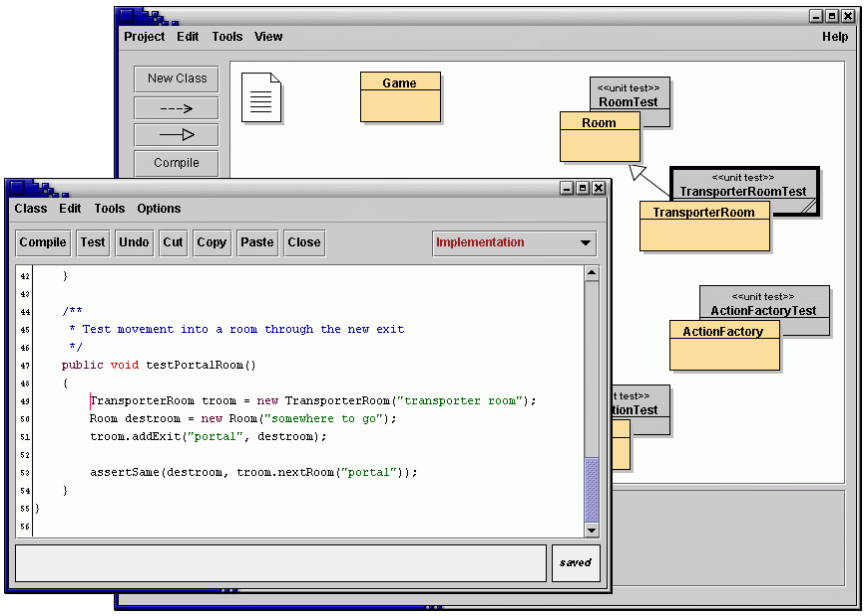

Figure 39 – Editing the TransporterRoomTest in the BlueJ editor.

The test runner dialog appears, showing the red bar indicating that a test has failed. Obviously, we have not implemented the `addExit()` method yet so we expected the test to fail. It is a good check however to make sure that there is nothing wrong with our test. If it had of succeeded without us implementing the method then we would need to be worried!

We edit the `TransporterRoom` class in order to implement the `addExit()` method. It turns out to be a simple matter of adding a pair to the `Map` of `Room` objects and exits. We add the implementation and run the tests again. We now get a green bar indicating that the test was a success. We can now move on to our next bit of functionality.

Whilst this walk through is a very simple example of how TDD works it shows the basic pattern of TDD. We write a test that will fail then implement code that makes it succeed. Then we start the cycle again.

The nature of TDD prevents the object interaction facilities of BlueJ being used, largely because the definitions of the objects and methods are not complete before

testing begins. However, there is nothing in BlueJ that prevents TDD from being used, and some facilities, such as being able to run the tests directly from the editor, actually help make some tasks easier.

5.6.2 Summary

The unit testing extensions to BlueJ aim to improve the tool support available for testing in introductory teaching. We have achieved this by integrating the JUnit testing framework into the BlueJ development environment in a manner that diminishes neither. At its most basic, the unit testing extensions allow the recognition and execution of JUnit test classes. We have extended this to also allow a JUnit test fixture to be moved onto the BlueJ object bench, and provided a method for converting objects on the BlueJ object bench into a JUnit test fixture. We have also developed a method for helping in the construction of unit test methods through the recording of object interactions on the object bench.

6 Chapter 6

IMPLEMENTATION

This chapter introduces some of the details of the implementation of the testing support in BlueJ. Firstly, the approach to serializing objects into test fixtures is discussed, including possible alternative implementations. This is followed by an examination of the modification of the BlueJ architecture to allow for the creation of test fixtures. Finally, the technique used for the construction of test methods is described.

6.1 Implementation environment

As mentioned previously, BlueJ is implemented in Java using the JDK compiler. Ant [Ant2002] is used as the build tool resulting in the ability to build the system on diverse platforms such as Solaris, Mac OS X and Windows. BlueJ can be run on any 1.3 compliant Java platform. The source for BlueJ is approximately 80,000 lines of code.

The BlueJ development team consists of 3 programmers. All of the development of the testing support in BlueJ was the work of the author.

6.2 High level overview

The BlueJ system consists of 350 classes contained within twenty packages. A simplified diagram of the major components of BlueJ is shown in Figure 40. The architecture of BlueJ consists of two virtual machines which cooperate to provide facilities such as object interaction and debugging.

The top half of the diagram shows the classes that operate on the local virtual machine (the first virtual machine that is started when BlueJ is launched). This virtual machine is responsible for all aspects of the user interface, parsing the source for dependencies, and invoking the compiler. The bottom half of the diagram shows the classes that operate on the debug virtual machine (the virtual machine

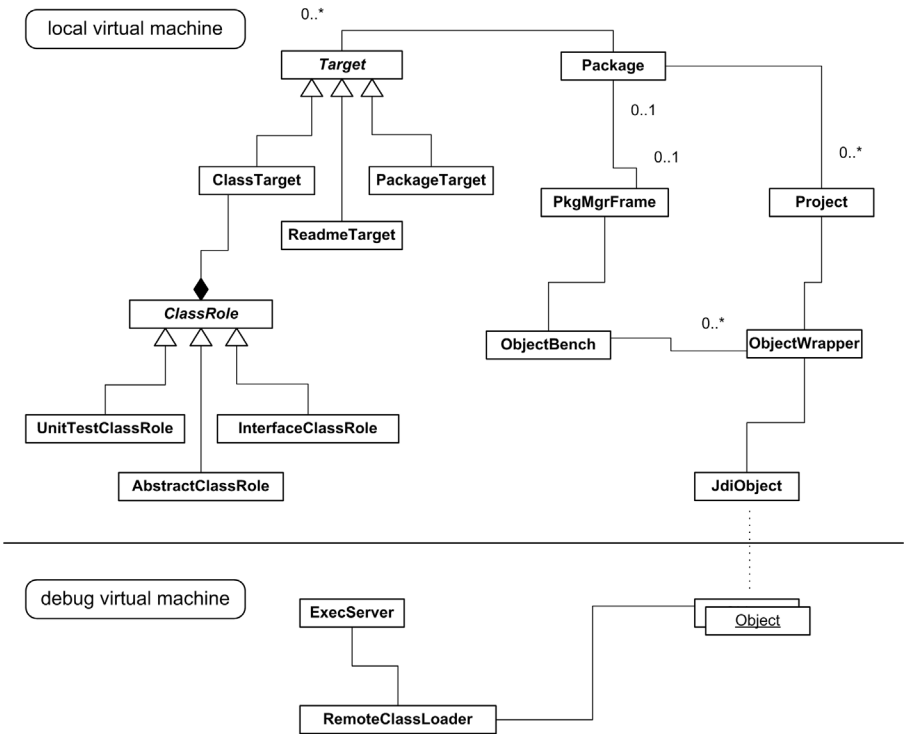

Figure 40 – A simplified view of the BlueJ system.

that is launched after start up by BlueJ). The debug virtual machine maintains the actual instances of objects on the object bench and the classes that allow operations on these instances. When a method is executed on an object through object interaction, the thread that runs the method lives on this debug virtual machine. All interaction with the debug virtual machine is performed using the JDI debugger interface [Javasoft2002].

The key user interface class is the `PkgMgrFrame`. This class implements the window and menus that are presented to the user. Within this window are two large panels. One panel displays objects on the object bench and is implemented by the class `ObjectBench`. Each object on the `ObjectBench` is represented by an `ObjectWrapper` which encapsulates the reference to the actual object on the debug virtual machine.

The other panel is a simplified UML diagram of a package and is implemented by the class `Package`. Each `Package` object corresponds one-to-one with an actual Java package. A `Package` object can exist outside of a `PkgMgrFrame` (in fact, the package objects for a project are all created on project load). When a `Package` object is placed into a `PkgMgrFrame` it then becomes visible. There are no references from `Package` objects to the `PkgMgrFrame` object that they are in. When a `Package` wants to indicate that the user has performed some action it raises a `PackageEvent` object which can be heard by any `PkgMgrFrame` objects that are registered as listeners. Without this decoupling, `Package` and `PkgMgrFrame` become so intertwined that the dependencies between them are hard to monitor. The decoupling will allow the `Package` code to be reused to display Java packages in a proposed class library browser within BlueJ.

The `Project` class is responsible for maintaining the collection of `Package` objects of a single project. It also controls all functionality that operates over an entire project, such as documentation generation.

The implementation of testing support presented four major problems

- Objects on the BlueJ object bench needed to be stored so that they do not need to be recreated every time the source is changed. This problem comes about because the behaviour of the objects in the presence of multiple versions of their classes is unpredictable;

- A technique was needed for recording BlueJ interactions and constructing unit test methods from them;

- Architectural changes were needed to the organization of the BlueJ virtual machines to better support dealing with JUnit classes; and

- JUnit's `TestRunner` interface had to be split in order to run JUnit tests across two cooperating virtual machines.

In the rest of this chapter, each of these implementation problems and the solution devised is presented.

6.3 Constructing test fixtures

The motivation for the initial implementation work in the testing area was not in fact testing. A common complaint about BlueJ is that objects on the object bench are removed on compilation. This removal of all live objects is required because their behaviour when interacting with objects that were constructed from previous versions of the source is unpredictable. There is a similar requirement in relation to testing where we want to be able to create test objects that survive recompilation.

The first approach to dealing with this problem was to consider analysing the changes made to the source to identify those changes that had no effect on the current set of objects. Whilst this works in the most trivial of cases (a class is edited with no dependency on any class depended on by any objects on the object bench) the analysis becomes complicated extremely quickly for more complex cases.

The second approach to the problem was to look at serializing objects on the object bench in some form. Serialization involves saving the contents of the object such that it can be reconstructed later in the same state. There are multiple techniques for serialization in Java, each with strengths and weaknesses. When evaluating their appropriateness for BlueJ the following criteria were considered

1. Ability to survive class evolution

   The objects being serialized are most probably going to be those whose source code is being worked on. It is important that the serialization is robust in handling changes to the source of the class;

2. Works without programmer intervention

   The serialization should work without the user of BlueJ needing to modify the class' source;

3. Works on many types of objects

It is important that as many types of object as possible are able to be serialized; and

4. Does not need to introduce advanced concepts
   Because BlueJ is an introductory teaching environment the serialization should not require the user of BlueJ to learn any advanced Java concepts.

The criteria were not absolute – support for all of them was not a requirement for use in BlueJ. As with all engineering decisions, we were looking for a solution which balanced the competing criteria. The following sections discuss a number of techniques for object serialization and consider their advantages and disadvantages.

6.3.1 Java Object Serialization (JOS)

JOS depends on programmers tagging a class as implementing `Serializable`. The `ObjectOutputStream` class can then be used to marshal all the fields of any objects of this class into a stream. The serializing technique is of course recursive, so any field which is of a class that is also marked as serializable is marshalled into the stream as well (in fact the serialization will throw an Exception if it encounters any fields in the object graph that are not serializable and which are not flagged with the `transient` keyword). Although the primitive Java types are not real objects and hence cannot implement `Serializable`, JOS has provisions to allow them to be marshalled into the output stream. Because it requires tagging classes with the `Serializable` tag, JOS requires programmers to make changes to source in order to support serialization. However, the change is minor and many of the standard Java classes support serialization so data structures such as members of the collections framework can be serialized with this technique.

Java serialization can handle evolution of classes but with some caveats. A compatible evolutionary change is one in which objects can be evolved from an old version of the class to a new version of the class and *vice-versa*. The serialization specification defines the following changes as valid

1. Adding fields

   It is allowable to evolve to a class that has an extra field because the extra field can be trivially set to the default value.

2. Adding classes

   By comparing the evolved class's hierarchy with the class hierarchy represented in the stream, additional classes can be detected and the additional class's fields can be initialised to the default values.

3. Removing classes

   In a method similar to adding classes, the serialization can detect when a class has been deleted from the evolved class's hierarchy. Because objects referenced in the deleted class may be referred to later in the stream the class still has to be demarshalled, but all primitive fields can be discarded and any objects of the deleted class demarshalled will be garbage collected if they turn out not to be referred to again.

4. Changing the access to a field

   When the access modifiers such as public, package, protected or private are changed it does not affect serialization. Serialization has special support from the VM that allows it to bypass the normal language field access rules.

5. Changing a field from static to non-static or transient to non-transient

   These fields are normally not serialized so changing a field to non-static or non-transient is the same as adding a field and the same technique is used to handle it.

The following changes to a class are considered incompatible. Some of these changes will work if evolution is only required from an old version to a new version.

1. Deleting fields

   When a field is deleted its value will not be written out to the stream. This is fine when evolving from an old to a new class but when going in the

reverse direction, the field data will not be in the stream and so the field will have to be initialised to a default value. This change is considered incompatible according to the JOS definition, but the evolution will work as long as the old version of the class can cope with the data in the deleted field being set to a default value.

2. Moving classes up or down the hierarchy
   The object data from each level in the hierarchy is serialized in order, so moving classes up or down the hierarchy means that data will not be available from the stream when required.

3. Changing a non-static field to static or a non-transient field to transient
   This is identical to deleting a field because marking the field transient or changing it to static means that it will now not be written out into the stream.

4. Changing the declared type of a primitive field
   The type of all primitive fields are serialized along with the data so if the field type is changed then the object can no longer be demarshalled because it will expect a different type.

JOS supports many of the forms of evolution that would be required by BlueJ. An important evolution that it does not support is renaming of fields (technically just a delete field and an add field but rename field needs to retain the value of the field). More importantly, in the cases where JOS does not work it fails without any means of correction. The serialized data is in a binary format and so it cannot be examined to make potentially simple corrections. For this reason, and because of the need for the source code to be tagged as `Serializable`, requiring modification of the source code, JOS is unsuitable for use as the serialization technique in BlueJ.

6.3.2 JSX

The "Java Serialization to XML" project (JSX) [Macmillan2002] utilises a similar approach to JOS, but rather than serializing to a custom binary stream format, it creates XML documents that represent the structure of the serialized objects.

The first advantage of JSX is that it does not require classes to be tagged as `Serializable`. This means that programmers do not need to change their source or learn about the 'implements' keyword before using the serialization.

The second advantage of JSX is that the output format is a human readable XML document. As a consequence of being able to view and edit the serialized data we can massage the data to perform more advanced class evolutions. For instance, a field rename can be performed by loading the serialized object into a text editor and editing the field name. Also, because JSX writes primitive types as strings into the XML document, it can change the primitive type of the field in the case where the string representation is convertible from one primitive to another.

However, performing class evolution like this is quite advanced – introductory students may not understand the XML format or understand how objects are structured when serialized. If students were to edit the object format and make a mistake (perhaps only renaming one instance of a field name in a collection of objects) then JSX will silently accept the error assuming that the unknown field name has been deleted.

It should be noted that some of the class evolutions possible because of the XML representation are not exclusive to JSX. It would also be possible to write a parser for the binary JOS format and to perform the same transformations of the serialized data that are feasible for JSX. However, the advantage of JSX is that by converting the objects to XML it leverages a lot of standard editing, parsing and tree transformation tools that are already available for the XML format.

```
<?xml version="1.0" encoding="UTF-8" ?>
<java version="1.4.0" class="java.beans.XMLDecoder">
  <void id="myController" property="owner"/>
  <object class="jaxax.swing.JPanel">
    <void method="add">
      <object id="button1" class="javax.swing.JButton">
        <string>Continue</string>
      </object>
    </void>
    <void method="add">
      <object class="javax.swing.JLabel">
        <void method="setLabelFor">
          <object idref="button1"/>
        </void>
      </object>
    </void>
  </object>
</java>
```

```
JPanel panel1 = new JPanel();
JButton button1 = new JButton("Continue");
JLabel label1 = new JLabel();
panel1.add(button1);
panel1.add(label1);
label1.setLabelFor(button1);
```

Figure 41 – A GUI component serialized to XML using XMLEncoder and how the component would look as Java code.

6.3.3 XMLEncoder and XMLDecoder

The introduction of the 1.4 JDK from Sun has seen a new form of serialization added to the Java platform. The new serialization technique was motivated by the desire to allow Java GUI components to be serialized into a form that was robust enough to survive class evolution of the GUI components. Because of the complexity of the implementation of the Java GUI components, coupled with the restrictions that JOS puts on evolution of classes, it had previously been impossible to save a GUI object in a state that would guarantee that they could be deserialized. The `XMLEncoder` and `XMLDecoder` classes have been added which can handle this serialization by converting a GUI component into an XML document. A sample of a serialized object is shown in Figure 41.

Unlike JOS or JSX, where each object's class can either rely on a default serialization routine or implement its own serialization technique by overriding `writeObject()`, the `XMLEncoder` uses a set of delegate classes which are responsible for the serialization of different types of classes. At its base level, `XMLEncoder` can work on a JavaBean component by using bean introspection to determine the properties of the class and for each property value (a property value is a special type of field that has both a getter and setter method) writing out the corresponding XML. The default persistence delegate class that is included with `XMLEncoder` handles this serialization of beans automatically.

It is also useful to be able to serialize objects that are not quite beans. Some of these objects like `Color` and `Font`, which do not have a no-argument constructor, can be serialized by providing a delegate that knows which bean properties should be passed to the constructor. Some objects which can only be constructed using a factory method can be serialized with `XMLEncoder` using persistence delegates that know which expression to output to create the object. It is important to note with `XMLEncoder` and `XMLDecoder` that any special case code required to serialize an object is only needed on the encoding side. The generated XML document is a complete description of all the constructor calls and method calls needed to recreate an object and hence the `XMLDecoder` does not need to be specialised for objects that are difficult to serialize.

`XMLEncoder` is not suitable for the serialization in BlueJ because it cannot deal with all types of objects. In particular, whilst it has support for serializing many of the complex standard Java GUI classes, it would require the user to add persistence delegates for any of their classes that did not conform to the JavaBean specifications. Requiring students to write classes in a JavaBean format is non-trivial and is not suitable for introductory teaching.

The notion of reconstructing objects as a sequence of constructor and method calls is an interesting idea however, and one that naturally fits in with the interaction

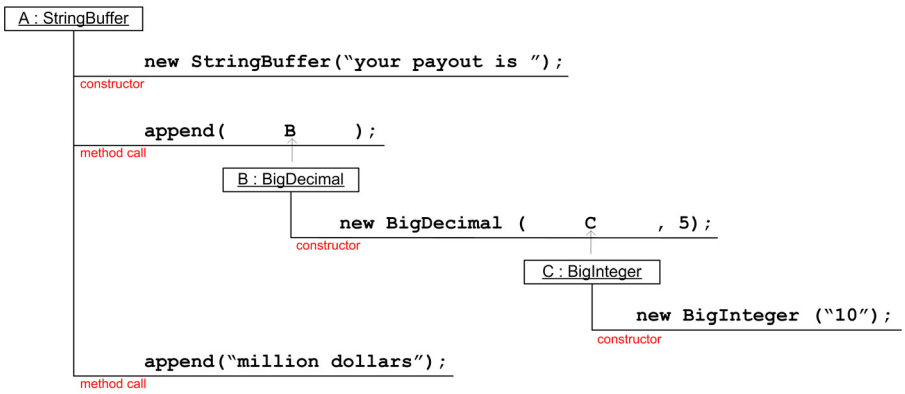

Figure 42 – A graph recording the transitive closure of all operations on the object A.

mechanisms already in place within BlueJ. This idea is investigated in the next section.

6.4 Creation of the text fixture and test methods

BlueJ has an advantage over other serialization methods in that it has the facility to record *how* an object came to be in its current state, not just what its state currently is. This is because objects placed on the object bench in BlueJ are constructed and manipulated through the BlueJ user interface. BlueJ can record each interaction with the user, be it an object construction or a method call, and use this to reconstruct the object at a later point.

The first model for implementing the creation of test fixtures in BlueJ was heavily influenced by traditional implementations of serialization. When serializing an object using traditional serialization, a transitive closure of the object (i.e. the object and all objects referenced by it) is formed and this closure is flattened into a data stream. We implemented a technique where the transitive closure of all operations on an object is formed and recorded by BlueJ as a graph. For any operation that requires another object as a parameter, the graph of the transitive closure of that object is linked in at that point. An example of the graph structure formed is shown in Figure 42.

The idea was that the user interface of BlueJ was going to allow an individual object to be turned into a fixture, through selection of a menu item, and at this point the graph of the operations on the object were to be turned into source and inserted into the test class. The ordering of operations was to be calculated from an analysis of the graph structure. A problem with this implementation is that care needs to be taken to correctly capture the state of objects on the object bench *at the time* that they are used as a parameter in another object's method call. For instance, consider the case where an object X is constructed taking a `StringBuffer` as a parameter. Let us imagine we have constructed the `StringBuffer` on the object bench as an object called S and then passed it as a parameter to the constructor call for X. We now call the `append()` method on S to add some characters to the `StringBuffer`. A naïve attempt at serializing X would perform the operations to construct S, including the `append()` method call, and then pass this object S as a parameter to the constructor of X. Clearly, the correct solution is to only perform operations on S, up unto the point at which it is used as a parameter, therefore not including the final `append()` method call. Whilst not an unsolvable problem, correctly dealing with situations like this complicates the data structures used to record operations.

Instead, we took a step back and looked again at the overall goals of the serialization. The purpose of the serialization is to construct a test fixture – objects of a known state that can be used by each test method without having to be reconstructed in the test method. In most cases there will be more than one object in a fixture. If we look at the fixture as a set of objects that need to be serialized, we can see that the fixture corresponds with the object bench. The object bench is itself a set of objects. Rather than trying to serialize a single object, the entire state of the object bench can be serialized, thereby constructing the fixture for a particular test case.

Once this observation had been made it was trivial to see a solution for recording the construction of the fixture. By considering the object bench as a whole, all operations, on *any object of the object bench* are recorded in the sequence that they

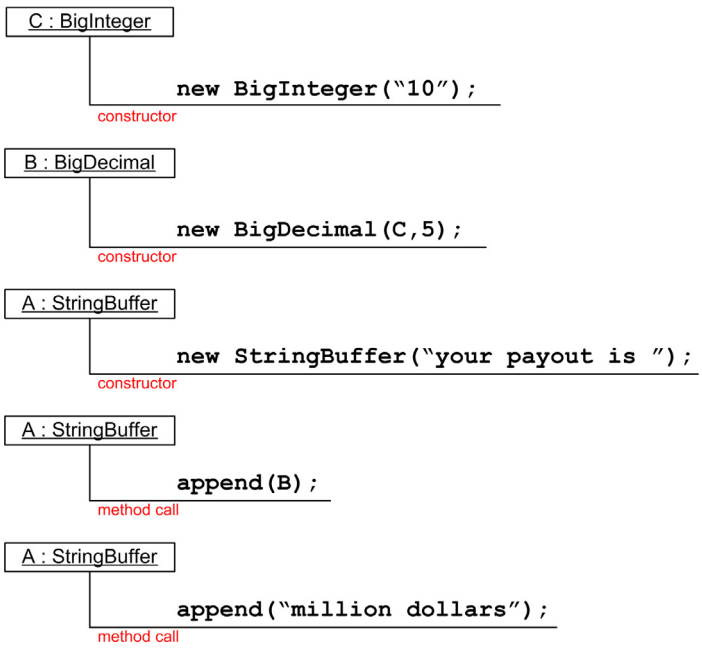

Figure 43 – The objects on an object bench recorded as a sequence of operations.

occur. To recreate the state of the object bench, the operations are replayed in the order in which they were recorded, assuming an identical starting state. Rather than storing operations in a complicated graph structure, they are recorded in a simple linked list (see Figure 43).

This implementation does have some caveats. Firstly, the fixtures for a test case must include all objects on an object bench – individual objects cannot be turned into fixtures. Secondly, all operations on all objects are included in the resulting source, even those that turn out to be redundant. For instance, if an object is created on the object bench and then removed from the object bench, the source code to construct it will be retained. This does not pose too much of a problem as it occurs rarely and the resulting redundant code is generally insignificant compared to the other set up code. It would be possible to scan the record of interactions for objects that are not used, and to then prune these from the list, however this has not been done in the current implementation.

The last problem is to ensure that the starting state for the replay of the object interactions is consistent. Because of the nature of the BlueJ object bench, where it must be cleared when classes are recompiled, the obvious starting state is an empty bench. Therefore, the recording of object interactions is always reset on compilation.

The actual construction of the test fixture set up code is performed by translating the recorded interactions into source code. Interactions that result in the construction of an object must be distinguished from other interactions in order for the field definitions of the objects to be inserted at the top of the test case. Both the existing field definitions and set up code must be replaced completely by the newly constructed code. A Java grammar using the ANTLR [Parr2002] parser generator was modified to identify the regions of the source code that need to be replaced by the newly constructed code.

In the same manner that BlueJ records the user interactions to create test fixtures, BlueJ also records the interactions and assertions that make up each test method. When a test method is being recorded and an interaction returns a result, BlueJ augments the result dialog with an assertion panel. If the user checks the checkbox indicating they wish to make an assertion, BlueJ translates the assertion into one of the standard JUnit assertion statements.

Because the variable "result" is used as the name of the return value of each method call, it is important to be able to either reuse the "result" variable or change the result name so that there is no naming clash. It was decided to structure the test methods in a way that the scoping rules of Java allow the "result" variable to be reused. Before each method call that is responsible for an assertion, BlueJ uses curly brackets to introduce a new scope. Within this scope, "result" is declared with the correct type and an assignment is made to it. Any assertions that have been made are then called within this scope. For arrays, where the method call must only be made once, yet multiple assertions can be made on the result, this scoping technique is particularly useful. This is shown in Figure 44.

```
{
  Parser parser_1 = new Parser();

  // assertion
  {
    String[] result = parser_1.tokenizeAndLower("");
    assertNotNull(result);
  }

  // normal method interactions
  parser_1.addToken("y");
  parser_1.addToken("x");

  // an array assertion
  {
    String[] result = parser_1.tokenizeAndLower(" AA Ab  bb");
    assertEquals(result[0], "aa");
    assertEquals(result[1], "ab");
    assertEquals(result[2], "bb");

  … rest of test method
```

Figure 44 – Using the scoping rules of Java to allow the "result" variable to be reused within a method.

6.5 Architectural changes to support testing

Recall that the basic architecture of BlueJ is two virtual machines – one that supports the user interface and compilation (called the local virtual machine), and one that supports the execution of methods and construction of objects (called the debug virtual machine). Communication between the two virtual machines is performed with the Java Debug Interface (JDI). JDI allows primitive types to be exchanged between machines but uses object references to facilitate the access to objects in one machine from the other. JDI does not therefore return a serialized object when returning an object from a method call, it returns only a reference to the object in the debug virtual machine.

The implementation of test fixtures required a substantial change to the architecture of the cross virtual machine communication. All communication between the virtual machines is routed through a virtual machine controlling class that handles the marshalling of parameters in the local virtual machine. The pre-unit testing implementation of this class dealt only with `String` objects, as this was the most flexible primitive type to deal with. Using `String` objects also avoided having to

deal with cross machine references. To deal with the construction of test fixtures and the execution of test methods required an interface that could handle object references as both parameters and return values.

The new implementation of the virtual machine controlling class supports both of these. When a method returns an object reference, BlueJ wraps this object reference in its own type called a `DebuggerObject`. `DebuggerObject` objects can then be placed on the object bench by constructing an `ObjectWrapper` and placing it as a component on the bench. The `ObjectWrapper` handles all the details of constructing the popup menus for the object's methods.

When the recording of a test method is begun, BlueJ needs to place the fixture objects on the object bench. To do this it needs to execute the `setUp()` method of the test case and place all the fields of the test case on the object bench. The code which executes the `setUp()` method on the debug virtual machine returns a `List` of objects that have been constructed. The reference to the `List` object is returned and this is then converted into an array of `DebuggerObject` objects that are then placed on the object bench.

The other architectural change that was required was changes to the security system of the debug virtual machine in order to support the execution of the test case `setUp()` method. Because the `setUp()` method is a protected method, it cannot be executed by classes that do not directly inherit from it. However, Java has the ability to suppress the access checks on a method by installing a custom security manager in the virtual machine. Once the access checks are suppressed, the method must also be marked as accessible by calling its `setAccessible()` method. The suppression of access checks in the debug virtual machine would appear to perhaps allow students to construct code that may bypass the standard access controls of Java. This is not the case though, because the compiler still will not allow code to be compiled that violates the Java access rules. The only way that students may encounter the changes to the security system is if they use reflection to gain access to methods. Additionally, methods and fields obtained through reflection must still

have their `setAccessible()` method called before all access checks are suppressed. It is unlikely that any student will *inadvertently* encounter this difference between the BlueJ virtual machine and a standard virtual machine.

6.6 Implementing the Runner

The JUnit framework includes two user interfaces that allow tests to be run. The `TextRunner` executes a test case and displays the result to standard output. The `SwingRunner` interfaces displays the result of the test in a GUI. The basic `SwingRunner` interface had to be modified to work with BlueJ's dual virtual machine architecture. The simplest implementation would have been to run test cases on the local virtual machine, thereby using an almost unchanged `SwingRunner` interface. However, all the other method execution and object construction performed in BlueJ is done on the debug virtual machine. Executing the code on the local machine would potentially result in differing behaviour between test creation (when the code would run on the debug machine) and test execution (when the code would run on the local machine).

Another possibility is for the entire `SwingRunner` interface to be loaded into the debug virtual machine, thereby allowing it to run tests on the desired virtual machine. Unfortunately, the threads running in the debug virtual machine are started and stopped numerous times in order to perform some of the method interactions. If the user interface code were running on this machine it would become unresponsive whilst its graphics helper thread was stopped.

The actual implementation splits the `SwingRunner` code into two parts. One part that deals with the actual execution of tests resides on the debug virtual machine. The other part of the code is passed the test class, and after executing the test case, returns a `TestResult` object. The runner user interface on the local virtual machine accesses the `TestResult` object and displays the results.

6.7 Summary

The integration of testing support in BlueJ posed numerous implementation challenges. Foremost was the challenge of determining a method for serializing objects on BlueJ's object bench into test fixtures. The final method selected was to have the fixture creation code "replay" all the BlueJ interaction events used to create objects on an object bench. A similar technique is used to create the unit test methods. Other implementation changes were mandated by the BlueJ architecture of having two separate virtual machines.

7 Chapter 7

STATUS AND FUTURE WORK

In this chapter the status of the work presented in this thesis is discussed along with discussion of ideas for the future direction of work relating to BlueJ.

7.1 Status

Although the design of the refactoring functionality described in chapter 3 has been completed, only the basic back-end of the functionality has been implemented. Other than prototype mock interfaces, no implementation has been made of the interface. A prototype implementation is planned for the future.

A working version of the unit testing functionality described in chapter 5 has been developed and is currently undergoing testing. Release of the production version is awaiting the completion of several tasks:

- Development of user manual.
  Whilst the aim of adding unit testing to BlueJ was to integrate functionality as simply as possible, it is evident that the functionality requires an explanation for students to be able to use it to full effect. The development of a simple tutorial document (either as an extension to the current BlueJ tutorial or as a separate document) will greatly assist teachers introducing the functionality to students.

- Internationalization of user interface strings.
  Because BlueJ is used in many different countries it supports the internationalization of all user interface text. For the unit testing extension, adding the new text translations will not be possible for all currently supported languages, but is necessary for the major languages that we support such as German and Chinese.

- Integration with the latest BlueJ release.

  Whilst the unit testing extension was under development, other work on BlueJ did not cease. Two versions of BlueJ, including the major release 1.2.0, have been completed in the interim and work is required to integrate the unit testing functionality into the main source tree.

It should be noted that BlueJ is a being used in production environments in universities around the world and hence stability and robustness are key criteria for the inclusion of new features. Because of this, the unit testing extension has been released to the public firstly in the form of a beta version, and this has lead to some valuable feedback being received about the extension.

7.2 Usability Study of the Unit Testing Extension

An initial usability study was conducted to determine the effectiveness of the interface at performing some representative student tasks [Kantner1994]. The usability study used four participants [Nielsen1994], two first year undergraduates, one latter year undergraduate and one postgraduate. Because the aim of the study was to identify usability faults with the program, it was considered useful to obtain usability feedback from a wider variety of participants than just introductory software engineering students.

7.2.1 Experimental Procedure

Participants were invited to take part in the study through the posting of notices in computing laboratories. The invitation was made to students who were already familiar with the BlueJ development environment as we wished to concentrate on the usability aspects of just the unit testing extension.

Participants in the test were given a consent form and a leaflet describing the project and the aims of the usability test. At the start of the allotted testing period (one hour was nominally set aside for each participant), the participants were given a printed tutorial on the unit testing extension in BlueJ. They were allowed to refer to this during the testing, although most of them chose not to.

The usability test undertaken was a think-aloud test. In this form of testing, the participant is asked to perform a set of tasks whilst articulating the thought processes that they are going through to achieve each task. For instance, whilst performing the task "open a project X", they first might go to the menu bar and click in the "File" menu, whilst saying aloud "I am going to the File menu to see if there is a menu item to open a project". The usability sessions are audio taped and the session is analysed to find places where the mental model that a participant had for a particular task did not match the actual interface of the program.

A number of sample projects were created (at various stages of completion) and some representative tasks on the projects were developed. As the participant stepped through the tasks, the usability problems that they identified were recorded.

The summary of the 11 task that the users were asked to perform is presented in the following table. Obviously, the task sheet that was given to the participant was more detailed than this (containing details of each task such as where the project is located on disk and what the task involved) but these details have been stripped out of the table presented here.

Task No.	Task
1	Recognising existing unit tests
2	Running a single test method on an existing unit test class
3	Running all the tests on an existing unit test class
4	Running all the existing unit tests in a project
5	Interpreting the result of a unit test execution
6	Using test results to fix some failing Java source
7	Using object interaction on the object bench
8	Constructing a unit test for an existing class
9	Constructing a unit test method using object interaction
10	Constructing a fixture
11	Using a fixture in the construction of a unit test method

7.2.2 Results

A usability problem was noted if any of the participants failed to complete the task correctly or could not see how to proceed towards the completion of the task. Other problems were noted where the participant suggested that they would have expected a different user interface at a particular point, even if they then successfully completed the task. A final class of usability problem occurred when the participant discovered a bug in the software that put them in a state from which they could then not complete the task. Most of these bugs have since been dealt with but we have noted the problems here for completeness.

We have put the usability problems into 5 categories

- STATE

  The user is confused as to the state that the program is in. Additional visual indicators or dialogs may be needed to remind the user what state they are in. Alternatively, a redesign of the interface to remove stateful operations may be required;

- TEXT

  The terminology used in the user interface has confused the user. These problems can be addressed by reconsidering the wording used in menu items and dialogs;

- VISUAL

  The appearance of a user interface element was not clear. A redesign of the particular user interface element may be required;

- GENERAL

  There is some general usability flaw in the program. These need to be fixed on a case-by-case basis; and

- BUG

  The user has uncovered a bug in the unit testing extension.

Problem No.	Usability Problem	Category
1	User not clear that they are in a "recording" state.	STATE
2	Problems with use of "fixture" terminology. The user was unclear as to what constituted a "fixture", despite having some proficiency with JUnit.	TEXT
3	Construction of objects before using "create test method" menu item leaves objects on the bench but not recorded in the test.	BUG
4	Attempted to "run test" before ending the recording of a test.	STATE
5	Used "fixture to bench" before "create test method" because that's what the user wanted. Was unaware that "create test method" will do this automatically.	GENERAL
6	Did not realise object interaction could be used to construct tests and wanted to construct tests by hand.	GENERAL
7	Use of a string in the assertion panel without quotes causes an error in the resulting test method. The resulting test class does not compile but no error message is shown to the user. Instead, the test class remains "striped".	GENERAL
8	User was unaware that clicking objects on the object-bench can be used to insert their name into method parameters.	GENERAL
9	Created a test named "balanceTest" as opposed to the JUnit naming standard "testBalance". This results in a test called "testBalanceTest".	GENERAL

10	Test failure dialog allows user to go to the line of the "test" class that fails. User was confused as to how to find out how this then relates back to the method that is being tested.	GENERAL
11	JUnit error output format was confusing when showing two strings that had been used in an assertion (the actual result and the desired results are displayed surrounded by <> rather than normal string quotes).	VISUAL
12	Running an individual test that succeeds results in only a message being displayed in the status bar. This was not noticed by the user.	VISUAL
13	After running a single class test and then running all tests, the user did not notice the new test results displayed in the test window.	VISUAL
14	User did not understand the significance of the JUnit green bar.	GENERAL

A usability success was noted when none of the participants had any difficulty accomplishing a task (or part of a task). The usability success table is not comprehensive (i.e. most of the tasks were completed successfully but are not listed here), instead it is a list of the areas that we had identified as potentially problematic, but which turned out not to cause any concerns.

Success No.	Usability Success
1	Recognised test classes by distinct colour and UML stereotype.
2	Technique for creating new test classes was clear.
3	Purpose of assertion panel was clear.
4	Automatic insertion of correct result into assertion panel was clear.

7.2.3 Discussion

The usability testing has uncovered some usability flaws in the unit testing extension, though none that we would consider serious enough to prevent the release of the unit testing extension to the public.

The main problem encountered was that the users did not realise that they were in a special "recording" state once they had selected "create test method". This is a problem because it could lead the user to fail to realise that some of the features of the unit testing extension even exist. We are evaluating methods that we can implement that make this change of state more evident to the user.

As part of the regular review cycle that all components of BlueJ undergo, along with feedback from users in the field, the usability study will help us identify and prioritize our work on improving the BlueJ unit testing extension.

7.3 Extended functionality

Some additional functionality considered in the initial design has not been implemented due to time constraints but may be included in the upcoming public release. Some of this functionality is described in the following sections.

7.3.1 Extending test methods

Whilst the test fixture of a test case can be extended by BlueJ, it is currently not possible to record a test method, use it in BlueJ and then continue the recording of the method. This functionality was not considered a priority due to the nature of test methods. Test methods are normally quite short and do not often require extending. Despite this, there are some situations where it may be useful and as the implementation is similar to the extension of a test fixture it should be easy to implement.

7.3.2 Support multiple test cases associated with a single target class

Currently, only one test case can be associated with each class in the BlueJ system. Whilst adequate for the majority of student use, some classes require multiple test

cases to help organise tests logically. Multiple test cases could be attached to a single class and stacked behind one another in the BlueJ class diagram.

7.3.3 Test coverage analysis

Test coverage tools provide metrics to understand the usefulness of test methods by analysing what proportion of the code in an application is reached by test code. Whilst full-scale test coverage analysis is beyond the scope of introductory students, a simple facility to highlight source code in a class that is not reached by test code would be useful.

7.4 Further tool support for introductory software engineering education

The analysis of the SWE-BOK in chapter 2 helped us identify two areas that we considered particularly deficient in tool support for introductory students, testing and refactoring. It also enabled us to review other areas of software engineering to gain an understanding of the current state of introductory tool support in those areas. Whilst an excellent basis for categorising and identifying areas of software engineering that may need tool support, the SWE-BOK is by no means a complete listing of all things that a software engineer may need to know. In particular, as the SWE-BOK is a document intending to capture the current mainstream areas of software engineering, there is the potential for other areas not mentioned to also make use of introductory tool support. Some of these areas that we see potential for adding introductory tool support are:

- Web services – the increasing interest in web based programming, and in particular the construction of web services means that these server environments may be considered for introductory software engineering courses. A tool that helps with server side deployment and deals with the issues of debugging and configuring server side programs may be useful.

- Version management – the change to object-oriented languages means that it is much more feasible that students will work on group projects, even in introductory courses. A tool that facilitates this group work through an

introductory version management system would also be useful for teaching the concepts of version control (branching, merging, conflicts, locking etc).

- Tools to support agile methodologies – unit testing is but one part of the agile methodologies that are becoming popular, even at an introductory level. There may be some benefit to tool support for some of these agile processes.

8 Chapter 8

CONCLUSION

This thesis has looked at the level of tool support for teaching introductory software engineering. The area of software engineering that we have examined in depth is the area of software product engineering. This area involves the design, coding, testing and maintenance of computer programs. In particular, we are interested in computer programs that are written in object-oriented languages and designed with object-oriented design techniques. Object-oriented languages are increasingly being used as a first language and we contend that many of the changes that object-orientation brings require more tool support than with procedural languages.

From the Software Engineering Body of Knowledge document, which describes concepts within software product engineering, we have composed a list of tasks that cover a large proportion of the practical skills that a first year student may be expected to develop from introductory courses. These skills represent such activities as entering code, building programs, designing programs and testing. It was our contention that some of these tasks are not well supported by software tools.

Software tools can play one of two roles in supporting a software engineering task. They may either be an integral part of the task, in the way that a compiler is an integral software tool when building programs. Alternatively, they may provide a level of pedagogical support for concepts, in the way that a development environment can reinforce important object-oriented notions such as objects and classes.

This thesis has made a number of contributions to the task of teaching software engineering concepts to introductory students. These include:

- A number of software engineering concepts that are suitable for inclusion in an introductory software engineering course have been identified.

- A number of software tools have been evaluated against their suitability to support these software engineering concepts in a teaching situation.

- Two areas we believe profit most from enhanced tool support in introductory courses have been identified: refactoring and testing

- A set of concrete refactorings suitable for inclusion in an introductory software engineering education has been identified.

- A detailed design for a refactoring tool suitable for use by first year students has been presented. The design presented reinforces the level at which refactorings operate. For instance, refactorings that act on methods are accessed through the "method" user interface component. Class level refactorings are accessed through a similar "class" user interface component. The design for a system wide "undo" stack that tracks large scale operations on the source code was also presented. The refactoring tool design has been integrated into an environment currently in widespread use in introductory courses to facilitate its implementation and adoption in the future. An implementation of this design has started.

- A detailed design of a testing facility suitable to teach modern testing activities to first year students has been presented. This design integrates the two leading software systems for introductory teaching of object-oriented programming and unit testing, BlueJ and JUnit, creating a new user interface style to approach testing activities. This new interaction style facilitated by our system allows techniques of testing (and the teaching of testing) that were not previously available to teachers and students.

- A full implementation of this testing facility has been developed, which has reached final testing stage and will be included in a full release version of the BlueJ environment.

We are convinced that the teaching of software engineering concepts in introductory courses has gained in importance over the last few years, and will continue the gain importance for the foreseeable future. For this development to be successful, the teaching community needs to develop teaching strategies and support tools geared towards this subject area. The contributions of this thesis are a step in that direction.

GLOSSARY

assets	terminology used by TestMentor – see *test fixture*
black-box testing	see *functional testing*
functional testing	testing that select test methods based solely on the public external interfaces of the source code being tested
integration testing	testing that tests the interactions between several smaller modules of code
structural testing	testing that selects inputs based on knowledge of the internal structure of the source code or its data structures
system testing	testing that tests the overall functionality of a complete system
test case	a class that holds a set of test methods. Each method with a name starting with test goes to make the set of tests for this test case
test class	see *test case*
test fixture	a common set of test data and collaborating objects shared by many test methods. Generally they are implemented as instance variables in a test case and are constructed in the setUp() method of the test case
test method	an single method containing test assertions that exists in a test case class
unit testing	testing that concentrates on a single module of code in a program
white-box testing	see *structural testing*

zork	from the zork manual… “Zork is a game of adventure, danger and low cunning. In it you will explore some of the most amazing territory ever seen by mortals. No computer should be without one!"

REFERENCES

[ACM1968] W. F. Atchison, S. D. Conte, J. W. Hamblen, T. E. Hull, T. A. Keenan, W. B. Kehl, E. J. McCluskey, S. O. Navarro, W. C. Rheinboldt, E. J. Schweppe, W. Viavant, and D. M. Young, "Curriculum 68: Recommendations for academic programs in computer science: a report of the ACM curriculum committee on computer science," in *Communications of the ACM*, vol. 11(3), 1968, pp. 151-197.

[ACM1979] R. H. Austing, B. H. Barnes, D. T. Bonnette, G. L. Engel, and G. Stokes, "Curriculum '78: recommendations for the undergraduate program in computer science - a report of the ACM curriculum committee on computer science," in *Communications of the ACM*, vol. 22(3), 1979, pp. 147-166.

[ACM1991] A. B. Tucker, B. H. Barnes, R. M. Aiken, K. Barker, K. B. Bruce, J. T. Cain, S. E. Conry, G. L. Engel, R. G. Epstein, D. K. Lidtke, M. C. Mulder, J. B. Rogers, E. H. Spafford, and A. J. Turner, *Computing Curricula '91*: ACM/IEEE-CS, 1991.

[ACM2001] ACM, "The Joint Task Force on Computing Curricula: Computing curricula 2001," *Journal of Educational Resources in Computing (JERIC)*, vol. 1, 2001.

[Allen2001] E. Allen, R. Cartwright, and B. Stoler, "DrJava: A lightweight pedagogic environment for Java," presented at Proceedings of the 32nd Annual SIGCSE Technical Symposium on Computer Science Education, Charlotte, NC, 2001.

[Allen2003] E. Allen, R. Cartwright, and C. Reis, "Production Programming in the Classroom," presented at Proceedings of the 34nd Annual SIGCSE Technical Symposium on Computer Science Education, Reno, NV, 2003.

[Ant2002] "Apache ANT Project," 2002, http://jakarta.apache.org/ant (accessed July 2002)

[Bagert1999] D. Bagert, T. Hilburn, G. Hislop, M. Lutz, M. McCracken, and S. Mengel, "Guidelines for Software Engineering Education Version 1.0," CMU, Technical Report CMUSEI-99-TR-032, October 1999.

[Barbey1994] S. Barbey and A. Strohmeier, "The Problematics of Testing Object-Oriented Software," presented at The Second Conference on Software Quality Management, Edinburgh, Scotland, 1994.

[Beck1989] K. Beck and W. Cunningham, "A Laboratory for Teaching Object-Oriented Thinking," presented at Object-Oriented Programming Systems, Languages and Applications (OOPSLA), New Orleans, LA, 1989.

[Beck1997] K. Beck, "Make it Run, Make it Right: Design Through Refactoring," in *The Smalltalk Report*, vol. 6, 1997, pp. 19-24.

[Beck1999] K. Beck, *eXtreme Programming eXplained*: Addison-Wesley, 1999.

[Beck2002] K. Beck, *Test Driven Development: By Example*: Addison Wesley, 2002.

[Boggs1999] W. Boggs and M. Boggs, *Mastering UML with Rational Rose*: Sybex, 1999.

[Brown1998] W. H. Brown, R. C. Malveau, H. W. McCormick III, and T. J. Mowbray, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis.*: John Wiley & Sons, 1998.

[Bruce2001] K. B. Bruce, A. P. Danyluk, and T. P. Murtagh, "Event-driven Programming is Simple Enough for CS1," presented at ITiCSE, Canterbury, UK, 2001.

[Chang1995] B. W. Chang, D. Ungar, and R. B. Smith, "Getting Close to Objects: Object-Focused Programming Environments," in *Visual Object Oriented Programming*, M. Burnett, A. Goldberg, and T. Lewis, Eds.: Prentice-Hall, 1995, pp. 185-198.

[Christopher1993] W. A. Christopher, S. J. Procter, and T. E. Anderson, "The Nachos Instructional Operating System," *USENIX Winter*, pp. 481-488, 1993.

[Connell1996] M. Connell and T. Menzies, "Quality Metrics: Test Coverage Analysis for Smalltalk," presented at TOOLS Pacific, Melbourne, 1996.

[Cook1992] W. R. Cook, "Interfaces and Specifications for the Smalltalk-80 Collection Classes," *ACM SIGPLAN Notices*, vol. 27, pp. 1-15, 1992.

[Culwin1999] F. Culwin, "Object Imperatives," presented at Proceedings of the 30th SIGCSE Technical Symposium on Computer Science Education, New Orleans, LA, 1999.

[Dewhurst1987a] S. C. Dewhurst, "Object Representation of Scope During Translation," presented at Proceedings of the 1st European Conference on Object-Oriented Programming (ECOOP), Paris, France, 1987.

[Eaton2001] N. J. Eaton, *Microsoft Visio Version 2002 Inside Out*: Microsoft Press, 2001.

[Fekete2000] A. Fekete, J. Kay, J. Kingston, and K. Wimalaratne, "Supporting reflection in introductory computer science," presented at Proceedings of the 31st SIGCSE Technical Symposium on Computer Science Education, Austin, TX, 2000.

[Ferret2002] L. Ferrett and J. Offutt, "An Empirical Comparison of Modularity of Procedural and Object-oriented Software," presented at Thirteenth International Conference on Engineering of Complex Computer Software, Annapolis, MD, 2002.

[Florijn2002] G. Florijn, "RevJava - Design Critiques and Architectural Conformance Checking for Java Software," 2002, http://www.serc.nl/people/florijn/papers/RevJava-overview-recent.pdf (accessed September 2002)

[Fowler1997] M. Fowler and K. Scott, *UML Distilled: Applying the Standard Object Modeling Language*: Addison-Wesley, 1997.

[Fowler1999] M. Fowler, *Refactoring: Improving the Design of Existing Code*: Addison-Wesley, 1999.

[Fowler2002] M. Fowler, "Catalogue of Refactorings," 2002, http://www.refactoring.com/catalog (accessed August 2002)

[Gamma1995] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*: Addison-Wesley, 1995.

[Ginat2001] D. Ginat, "Early Algorithm Efficiency with Design Patterns," *Journal of Computer Science Education*, vol. 11(2), pp. 89-109, 2001.

[Gold1991] E. Gold and M. B. Rosson, "Portia: an instance-centered environment for Smalltalk," presented at Object-Oriented Programming Systems, Languages and Applications (OOPSLA), Phoenix, AZ, 1991.

[Goldwasser2002] M. Goldwasser, "A Gimmick to Integrate Software Testing Throughout the Curriculum," presented at Proceedings of the 33rd Annual SIGCSE Technical Symposium on Computer Science Education, 2002.

[Gosling1999] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification, Second Edition*: Addison-Wesley, 1999.

[GPL1991] "The GNU General Public License," 1991, http://www.gnu.org/copyleft/gpl.html (accessed January 2001)

[Hetzel1988] W. Hetzel, *The Complete Guide to Software Testing*, 2nd ed. Wellesley, Mass.: QED Information Sciences, 1988.

[Hilburn1997] T. Hilburn and M. Towhidnejad, "Integrating Personal Software Process (PSP) Across the Undergraduate Curriculum," presented at Proceedings of the 1997 Frontiers in Education Conference, 1997.

[Hilburn1999] T. B. Hilburn, I. Hirmanpour, S. Khajenoori, R. Turner, and A. Qasem, "A Software Engineering Body of Knowledge Version 1.0," CMU, Technical Report CMUSEI-99-TR-004, April 1999.

[Hilburn2000] T. Hilburn and M. Towhidnejad, "Software quality: A curriculum postscript?," presented at Proceedings of the 31st Annual SIGCSE Technical Symposium on Computer Science Education, Austin, TX, 2000.

[Hitchens1994] M. Hitchens, P. English, and F. Maroufi, "Melmoth a class library management system," presented at Technology of Object-Oriented Languages and Systems 15, 1994.

[Hughes2000] L. Hughes, "An Applied Approach to Teaching the Fundamentals of Operating Systems," *Journal of Computer Science Education*, vol. 10(1), pp. 1-23, 2000.

[IntelliJ2002] IntelliJ, "IntelliJ IDEA 2.6," 2002, http://www.intellij.com (accessed

[Jackson1997] U. Jackson, B. Manaris, and R. McCauley, "Strategies for effective integration of software engineering concepts and techniques into the undergraduate computer science curriculum," presented at Proceedings of the 28th Annual SIGCSE Technical Symposium on Computer Science Education, San Jose, CA, 1997.

[Jarc2000] D. J. Jarc, M. B. Feldman, and R. S. Heller, "Assessing the Benefits of Interactive Prediction Using Web-based Algorithm Animation Courseware," presented at Proceedings of the 31st Annual SIGCSE Technical Symposium on Computer Science Education, Austin, TX, 2000.

[Javasoft2002] Javasoft, "JPDA Architecture," 2002, http://java.sun.com/j2se/1.4/docs/guide/jpda/architecture.html (accessed June 2002)

[Jeffries2000] R. Jeffries, C. Hendrickson, A. Anderson, and K. Beck, *Extreme Programming Installed*: Addison-Wesley, 2000.

[Johansson2001] T. Johansson and M. Nordström, "Introducing OOConcepts with CRC-cards and BlueJ - a case study," presented at OOPSLA01 - Workshop on Pedagogies and Tools for Assimilating Object Oriented Concepts, Tampa Bay, FL, 2001.

[Jones2001] E. Jones, "An Experimental Approach to Incorporating Software Testing Into The Computer Science Curriculum," presented at 31st ASEE/IEEE Frontiers in Education Conference, Reno, NV, 2001.

[JUnit2002] "The JUnit Testing Framework," 2002, http://www.junit.org (accessed March 2002)

[Kantner1994] L. Kantner, "Techniques for Managing a Usability Test," *IEEE Transactions on Professional Communication*, vol. 37(3), pp. 143-148, 1994.

[Kay1994] D. Kay, T. Scott, P. Isaacson, and K. Reek, "Automated grading assistance for student programs," presented at Proceedings of the 25th Annual SIGSCE Technical Symposium on Computer Science Education, Phoenix, AZ, 1994.

[Kernighan1984] B. W. Kernighan and R. Pike, *The Unix Programming Environment*, 1984.

[Khwaja1993] A. Khwaja and J. Urban, "Syntax-directed editing environments: issues and features," presented at Proceedings of the 1993 ACM/SIGAPP Symposium on Applied Computing, Indianapolis, IN, 1993.

[Kirby1997] G. Kirby and R. Morrison, "OCB: An Object/Class Browser for Java," presented at 2nd International Workshop on Persistence and Java, Half Moon Bay, CA, 1997.

[Kölling1999] M. Kölling, "The Design of an Object-Oriented Environment and Language for Teaching," in *PhD*: Basser Department of Computer Science, University of Sydney, 1999.

[Kölling2001a] M. Kölling and J. Rosenberg, "BlueJ - The Hitch-Hikers Guide to Object Orientation," *Journal of Object Oriented Programming*, 2001.

[Kölling2001b] M. Kölling and J. Rosenberg, "Guidelines for Teaching Object Orientation with Java," presented at Proceedings of the 6th conference on Information Technology in Computer Science Education (ITiCSE 2001), Canterbury, UK, 2001.

[Lappo2002] P. Lappo, "No Pain, No XP - Observations on Teaching and Mentoring Extreme Programming to University Students," presented at XP2002, Caligari, Italy, 2002.

[Larus1997] J. R. Larus, "SPIM S20: A MIPS R2000 Simulator," Computer Sciences Department, University of Wisconsin, 1997.

[Macmillan2002] B. Macmillan, "Java Serialization to XML (JSX)," 2002, http://www.csse.monash.edu.au/~bren/JSX (accessed June 2002)

[McCauley1998] R. McCauley and U. Jackson, "Teaching Software Engineering Early - Experiences and Results," presented at Proceedings of the 1998 Frontiers in Education Conference, Tempe, AZ, 1998.

[McClelland2002] D. McClelland, *Photoshop 7 Bible*: John Wiley & Sons, 2002.

[McDonald2001] J. McDonald, "Why Is Software Project Management Difficult? And What That Implies for Teaching Software Project Management," *Journal of Computer Science Education*, vol. 11(1), pp. 55-71, 2001.

[McKim1996] J. McKim and M. L. Manns, "Teaching OT: A Breadth-first Versus a Depth-first Approach," presented at Proceedings of the Educator's Symposium in conjunction with OOPSLA, 1996.

[Meyer2001] B. Meyer, "EiffelStudio: A Guided Tour," ISE Technical Report TR-EI-68/GT, 2001.

[Mutchler1996] D. Mutchler and C. Laxer, "Using Multimedia and GUI Programming in CS1," presented at ITiCSE, Barcelona, Spain, 1996.

[Naps2000] T. L. Naps, J. R. Eagan, and L. L. Norton, "JHAVÉ -- An Environment to Actively Engage Students in Web-based Algorithm Visualizations," presented at SIGCSE, Austin, TX, 2000.

[Netbeans2002] Netbeans, "Netbeans JUnit Module," 2002, http://junit.netbeans.org (accessed June 2002)

[Nielsen1994] J. Nielsen, "Estimating the number of subjects needed for a thinking aloud test," *International Journal of Human-Computer Studies*, vol. 41(3), pp. 385-397, 1994.

[Nilsson2000] D. R. Nilsson, P. M. Jakab, B. Sarantakos, and R. A. Stinehour, *Enterprise development with VisualAge for Java, Version 3*: John Wiley & Sons, 2000.

[Pan2002] J. Pan, "Software Testing," 1999, http://www.cmu.edu (accessed March 2002)

[Parr2002] T. Parr, "ANTLR," 2002, http://www.antlr.org (accessed July 2002)

[Pintado1990] X. Pintado, "Selection and Exploration in an Object-Oriented Environment: The Affinity Browser," Centre Universitaire d'Informatique, University of Geneva, 1990.

[Plauger1992] P. J. Plauger, *The Standard C Library*: Prentice Hall, 1992.

[Postema2000] M. Postema, M. Dick, J. Miller, and S. Cuce, "Tool Support for Teaching the Personal Software Process," *Journal of Computer Science Education*, vol. 10(2), pp. 179-193, 2000.

[Power2000] J. F. Power and B. A. Malloy, "Symbol table construction and name lookup in ISO C++," presented at Technology of Object-Oriented Languages and Systems - Pacific, 2000.

[Ramsey1992] N. Ramsey and D. R. Hanson, "A retargetable debugger," presented at Proceedings of the Conference on Programming Language Design and Implementation, 1992.

[Ramsey1994] N. Ramsey, "Literate Programming Simplified," *IEEE Software*, vol. 11, pp. 97-105, 1994.

[Rasala2000] R. Rasala, "Toolkits in first year computer science: a pedagogical imperative," presented at Proceedings of the 31st SIGCSE Technical Symposium on Computer Science Education, Austin, TX, 2000.

[Rasala2001] R. Rasala, J. Raab, and V. K. Proulx, "Java power tools: model software for teaching object-oriented design," presented at Proceedings of the 32nd SIGCSE Technical Symposium on Computer Science Education, Charlotte, NC, 2001.

[Reges2002] S. Reges, "Can C# Replace Java in CS1 and CS2?," presented at ITiCSE, Aarhus, Denmark, 2002.

[Roberts1997] D. Roberts, J. Brant, and R. Johnson, "A Refactoring Tool for Smalltalk," *Theory and Practice of Object-Systems*, vol. 3, 1997.

[Roberts1999b] D. Roberts and J. Brant, "Refactoring Tools," in *Refactoring: Improving the Design of Existing Code*, M. Fowler, Ed.: Addison-Wesley, 1999, pp. 401-407.

[Rosenberg1997] J. Rosenberg and M. Kölling, "I/O Considered Harmful (At least for the first few weeks)," presented at Proceedings of the Second Australasian Conference on Computer Science Education, Melbourne, Australia, 1997.

[Schulz2000] D. Schulz and F. Mueller, "A Thread-Aware Debugger with an Open Interface," presented at International Symposium on Software Testing and Analysis, Portland, OR, 2000.

[Seemann1997] J. Seemann, "Extending the Sugiyama Algorithm for Drawing UML Class Diagrams: Towards Automatic Layout of Object-Oriented Software Diagrams," presented at Proceedings of Graph Drawing, 5th International Symposium, GD'97, Rome, Italy, 1997.

[Shaw1991] M. Shaw and J. Tomayko, "Models for undergraduate project courses in software engineering," presented at Proceedings of the Fifth Annual SEI Conference on Software Engineering, Pittsburgh, PA, 1991.

[Silvermark2002] Silvermark, "Test Mentor Java Edition User Reference 5.4," 2002, http://www.silvermark.com/documentation/stmjava.pdf (accessed February 2002)

[Spear1994] P. Spear, *Return to Zork - The Official Guide to the Great Underground Empire*: Brady Publishing, 1994.

[Stasko1993] J. Stasko, A. Badre, and C. Lewis, "Do Algorithm Animations Assist Learning? An Empirical Study and Analysis," presented at Proceedings of the INTERCHI '93 Conference on Human Factors in Computer Systems, Amsterdam, Netherlands, 1993.

[Talbott2001] N. Talbott, "Testing in reverse," 2001, http://www.rubyconf.org/2001/talks/testinginreverse (accessed March 2002)

[Transmogrify2001] "Transmogrify - A Java Refactoring Tool," 2001, http://transmogrify.sourceforge.net (accessed July 2001)

[Whittaker2000] J. Whittaker, "What is software testing? And why is it so hard?," *IEEE Software*, vol. 17, pp. 70-79, 2000.

[Zeller1996] A. Zeller and D. Lütkehaus, "DDD - A Free Graphical Front-End for UNIX Debuggers," *SIGPLAN Notices*, vol. 31, pp. 22-27, 1996.